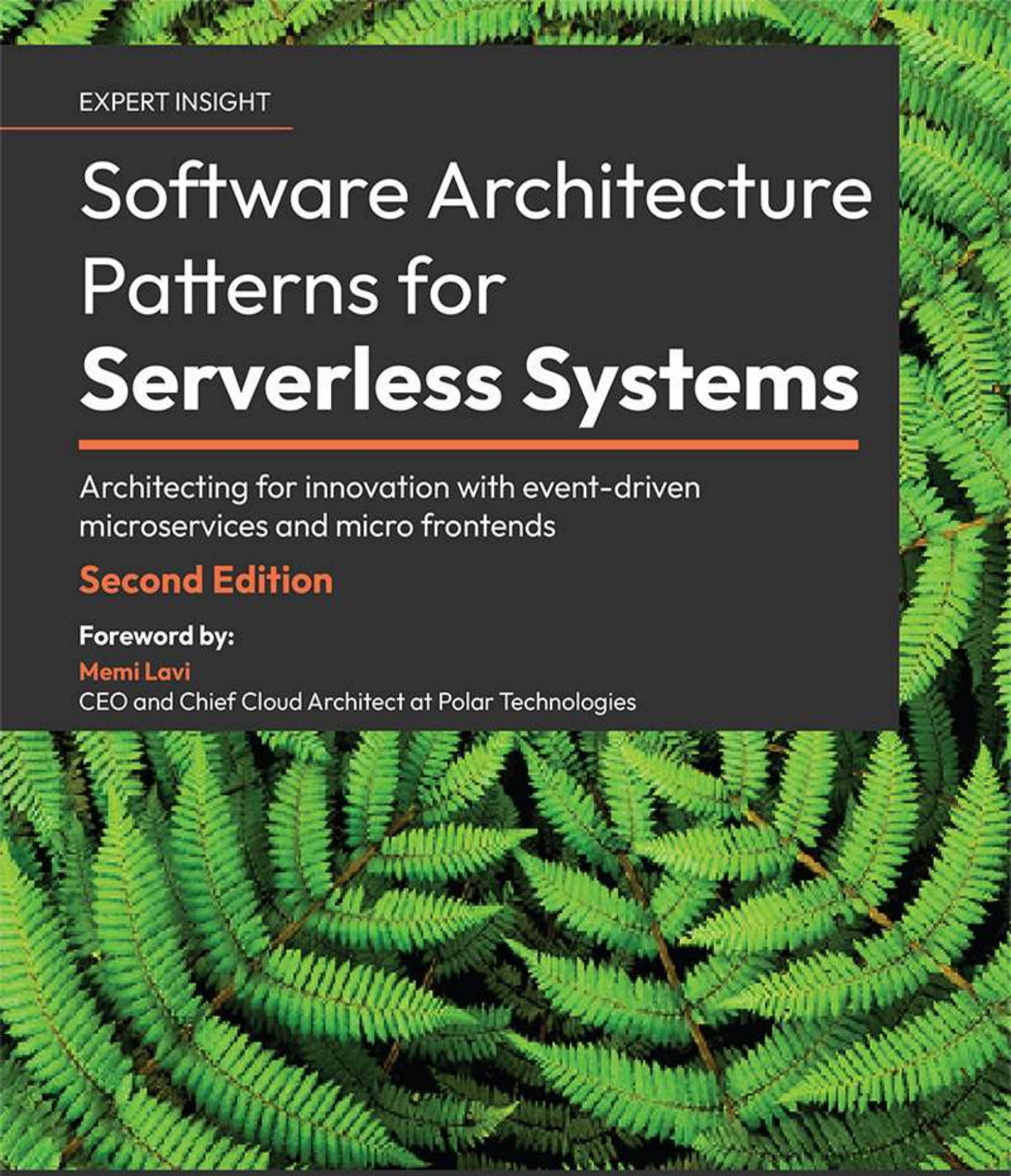


EXPERT INSIGHT

Software Architecture Patterns for Serverless Systems

Architecting for innovation with event-driven microservices and micro frontends

Second Edition

Foreword by:
Memi Lavi
CEO and Chief Cloud Architect at Polar Technologies

John Gilbert

packt

Software Architecture Patterns for Serverless Systems

Second Edition

Architecting for innovation with event-driven microservices and micro frontends

John Gilbert

‹packt›

BIRMINGHAM—MUMBAI

Software Architecture Patterns for Serverless Systems

Second Edition

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Senior Publishing Product Manager: Denim Pinto

Acquisition Editor – Peer Reviews: Saby Dsilva

Project Editor: Amisha Vathare

Content Development Editor: Shikha Parashar

Copy Editor: Safis Editing

Technical Editor: Aneri Patel

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Presentation Designer: Ganesh Bhadwalkar

Developer Relations Marketing Executive: Vipanshu Parashar

First published: June 2021

Second edition: February 2024

Production reference: 1230224

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul’s Square

Birmingham

B3 1RB, UK.

ISBN 9781803235448

www.packt.com

Foreword

Serverless.

This magic word has captured the imagination of countless cloud developers in recent years. What can be better than a serverless runtime, where you just slap your code into it, and it simply runs?

Well, as it turns out, having it “simply” run is not that simple. Serverless systems, while introducing new levels of simplicity, also brought with them several new architectural paradigms.

How do you secure serverless functions? How do you monitor them? How much is it going to cost? How does it affect the client code?

In fact, the serverless paradigm, which began as an IT-related paradigm, allowing running lightweight pieces of code on fully managed infrastructure, taking care of scaling, monitoring, updating, and more, formed into something much larger. How can we lightweightize (yep, that’s a word now) almost everything in our software? The microservices we used last year to make our software modular and lightweight suddenly became enormous chunks of code that can be further dismantled into lightweight functions, running on serverless platforms.

This led to some of the interesting developments in the software architecture field. For example, why should we aspire to run lightweight code in the backend, but stick to a massive monolith in the frontend? And because of this, micro frontends were born.

Why expose synchronous, strongly coupled web APIs, when we can use small, loosely coupled events instead?

All these questions led to a clear insight – a new software architecture paradigm is necessary, one that takes advantage of all these developments.

But how do you learn a new architecture? How can software architects, one that worked on dozens of microservices-based systems, become accustomed to the new paradigm and feel comfortable enough to try it on the new system they are working on?

Well, it turns out that there is more than one way of doing that.

In microservices, we had the seminal “Microservices” article by Martin Fowler that laid the foundations for what is now the most widely used architecture pattern out there. But even this article, quoted in every microservices book and lecture, is not a practical, walkthrough guide. After all, there are quite a few microservices books…

And for that, we have this book.

The book you hold in your hands (or maybe reads from a screen) is the ultimate one-stop-shop for the new architecture style, epitomized as “serverless.” It describes the new lightweight-oriented architecture that can be created using serverless cloud engines such as AWS Lambda, all the way to the frontend using micro frontends.

But it doesn’t stops there.

It also touches on the foundational topics such as Domain Driven Design and the SOLID principles, which form the basis of modern software design, explains the concept behind events, and also explains database-related patterns such as CQRS.

Finally, with the addition of micro frontends, you will be able to design your own serverless, event-driven systems that are flexible enough to allow

you to deliver value at the rate that your business requires. This was really just a small taste of what you can find in it.

In short – this is the perfect book for the new wave of software architecture patterns.

Reading it, and then visiting again the parts you're working on, will definitely make you a better software architect, fully adapted to the modern world of serverless architecture.

Memi Lavi CEO and Chief Cloud Architect at Polar Technologies

Contributors

About the author

John Gilbert is a CTO with over 30 years of experience in architecting and delivering software systems across many industries. His cloud journey has spanned all the levels of cloud maturity, from lift and shift and software-defined infrastructure to microservices and continuous deployment. He was an early serverless adopter and put his first serverless workloads into production just months after AWS Lambda's introduction. He has also authored *Cloud Native Development Patterns and Best Practices* and *JavaScript Cloud Native Development Cookbook*. He finds delivering serverless solutions to be most fun and satisfying, as they force us to reconsider how we reason about systems and enable us to accomplish far more with much less effort and risk.

To my wife, Sarah, and our children, for their endless love and support on this journey

About the reviewers

Marco Lenzo is an experienced technical leader who has designed and implemented multiple highly available distributed systems. Currently, he holds the position of Lead Architect for an **Infrastructure-as-a-Service** (**IaaS**) product developed by several agile teams.

His love for IT started at a young age and is not limited to software development alone, but it also encompasses security, systems, and networks. He is a strong supporter of Agile, DevOps, GitOps and cloud-native technologies. He was an early adopter of Kubernetes and holds the **Certified Administration** (**CKA**) and **Kubernetes Certified Security Specialist** (**CKS**) certifications.

Marco enjoys coaching and mentoring and is an active contributor to the software development community. He currently runs a blog and YouTube channel focusing on software architecture, DevOps, and leadership.

I wish to thank John and Packt for this great opportunity. It was a pleasure reviewing a book which leverages skillfully the pillars of software development, such as DDD and the SOLID principles, to deal with complexity of the cutting-edge architecture of serverless systems.

Re Alvarez Parmar is a Principal Specialist Solutions Architect at **Amazon Web Services** (**AWS**), where he advises Fortune-100 companies

on building cloud-native systems. He has over 20 years of experience with architecting, building, and operating enterprise systems. He currently focuses on modern application development patterns, cloud architecture, and Kubernetes.

Being completely self-taught, Re is passionate about knowledge sharing and believes in giving back to the community that has given him so much. His works have been featured on the AWS Containers and Architecture blogs. He has given talks on cloud-native architecture at KubeCon US, and has been a regular speaker at AWS Re:Invent for the past 6 years.

As I sit down to write these acknowledgments, I am filled with gratitude and humility, knowing that no book comes to fruition through the efforts of one person alone. Behind the pages you hold in your hands is a network of support, encouragement, and unwavering love that I am privileged to acknowledge.

First and foremost, I must express my deepest gratitude to my wife, Nadia. Throughout the arduous journey of editing this book, she was my steadfast anchor, providing unfaltering support and understanding. She selflessly managed our home and took on countless responsibilities so that I could dedicate long hours to this book. Without her, none of this would have been possible, and for that, I am eternally grateful.

To my children, Jayden, Norah, and Neil. Your immeasurable energy, curiosity, and creativity is a constant source of inspiration. And finally, my mother, whose unwavering support and belief in me have been a guiding light in my life.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/B8TubSxc35

Contents

- Preface
  - Who this book is for
  - What this book covers
  - To get the most out of this book
1. Architecting for Innovation
   - Continuously delivering business value
     - By the skin of our teeth
     - Through high-velocity teamwork
   - Dissecting lead time
     - Risk mitigation
     - Decision making
     - Software development life cycle methodology
     - Hardware provisioning
     - Deployments
     - Software structure
     - Testing and confidence
     - Dependencies and inter-team communication
   - Dissecting integration styles
     - Batch integration
     - Spaghetti integration
     - Real-time integration
     - Enterprise application integration
     - Shared database
     - Service-oriented architecture
     - Microservices
   - Enabling autonomous teams with autonomous services
     - Autonomous services – creating bulkheads
       - Asynchronous inter-service communication
       - Fortified boundaries
     - Event-first – valuing facts
       - Inversion of responsibility
       - Events as first-class citizens
       - Idempotence and ordered tolerance
     - Serverless-first – creating knowledge

        - Self-service
        - Disposable architecture
    - Data life cycle – fighting data gravity
    - Micro frontends – equalizing tiers
    - Observability – optimizing everything
    - Organic evolution – embracing change
- Summary

2. Defining Boundaries and Letting Go
- Learning the hard way
- Building on proven concepts
    - Domain-driven design
        - Bounded context
        - Domain aggregate
        - Domain event
    - SOLID principles
        - Single Responsibility Principle
        - Open-Closed Principle
        - Liskov Substitution Principle
        - Interface Segregation Principle
        - Dependency Inversion Principle
    - Hexagonal Architecture
        - Function-level (nano)
        - Service-level (micro)
        - Subsystem-level (macro)
- Thinking about events first
    - Start with event storming
    - Focus on verbs instead of nouns
    - Treat events as facts instead of ephemeral messages
    - Turn APIs inside out by treating events as contracts
    - Invert responsibility for invocation
    - Connect services through an event hub
- Dividing a system into autonomous subsystems
    - By actor
    - By business unit
    - By business capability
    - By data life cycle
    - By legacy system

- Creating subsystem bulkheads
  - Separate cloud accounts
  - External domain events
- Dissecting an autonomous subsystem
  - Context diagram
  - Micro frontend
  - Event hub
  - Autonomous service patterns
    - Backend For Frontend
    - External Service Gateways
    - Control services
- Dissecting an autonomous service
  - Repository
  - CI/CD pipeline and GitOps
  - Tests
  - Stack
  - Persistence
  - Trilateral API
    - Events
    - API Gateway
  - Functions
    - Nano architecture
    - Micro architecture
  - Shared libraries
- Governing without impeding
  - Providing automation and cross-cutting concerns
  - Promoting a culture of robustness
  - Harnessing the four key team metrics
- Summary

3. Taming the Presentation Tier
   - Presentation tier innovation – zigzagging through time
     - Client-side versus server-side rendering
     - Build-time versus runtime rendering
     - Web versus mobile
   - Breaking up the frontend monolith
     - By subsystem
     - By user activity

        - By device type
        - By version
    - Dissecting micro frontends
        - The main app
            - The index file
            - The importmap file
            - Micro-app registration
        - Micro-app
            - The entry file
            - Root component
        - Micro-app activation
            - Micro-app life cycle
            - Route-based activation
            - Manual activation
        - Mount points
        - Manifest deployer
        - Inter-application communication
    - Designing for offline-first
        - Transparency
            - Status indicators
            - Outbox
            - Inbox
        - Local cache
            - Caching code
            - Caching data reads
            - Caching data writes
        - Live updates
            - WebSocket
            - Long polling
    - Summary
4. Trusting Facts and Eventual Consistency
    - Living in an eventually consistent world
        - Staging
            - Cooperative
            - Atomic
        - Consistency
            - Transparency

        - Facts
        - Chain reaction
    - Concurrency and partitions
    - Order tolerance and idempotence
    - Parallelism
- Publishing to an event hub
    - Event bus
    - Domain events
        - Event envelope
        - Event-carried state transfer
        - Substitution
        - Internal versus external
    - Routing and channel topology
- Dissecting the Event Sourcing pattern
    - System-wide event sourcing
    - Event lake
        - Perpetual storage
        - Indexing events
        - Replaying events
- Event streams
    - Temporal storage
    - Stream-first event sourcing
    - Concurrency control
    - Micro-event stores
- Processing event streams
    - Micro batching
    - Choosing a programming paradigm
        - Imperative programming
        - Functional reactive programming
        - Stream processing
    - Creating a stream
    - Unit of work
    - Filtering and multiplexing
    - Mapping
    - Connectors
- Designing for failure
    - Backpressure and rate limiting

        - Poison events
        - Fault events
        - Resubmission
    - Optimizing throughput
        - Batch size function parameter
        - Asynchronous non-blocking I/O
        - Pipelines and multiplexing
            - Pipeline patterns
        - Sharding
        - Batching and grouping
            - Batching
            - Grouping
    - Summary
5. Turning the Cloud into the Database
    - Fighting data gravity
        - Competing demands
        - Insufficient capacity
        - Intractable volumes
    - Embracing the data life cycle
        - Create phase
        - Use phase
        - Analyze phase
        - Archive phase
    - Turning the database inside out
        - The transaction log
        - Derived data
    - Dissecting the CQRS pattern
        - System wide CQRS
        - Materialized views
        - Inbound bulkheads
        - Live cache
        - Capacity per reader, per query
    - Keeping data lean
        - Projections
        - Time to live
    - Implementing idempotence and order tolerance
        - Deterministic identifiers

        - Inverse optimistic locking
        - Immutable event triggers
    - Modeling data for operational performance
        - Nodes, edges, and aggregates
        - Sharding and partition keys
        - Single table design examples
            - Restaurant service
            - Customer service
            - Cart service
            - Delivery service
    - Leveraging change data capture
        - Database-first event sourcing
        - Soft deletes
        - Latching
    - Summary
6. A Best Friend for the Frontend
    - Focusing on user activities
        - A BFF service is responsible for a single user activity
        - A BFF service is owned by the frontend team
        - A BFF service is decoupled, autonomous, and resilient
    - Dissecting the Backend for Frontend pattern
        - Datastore
        - API gateway
        - Query and command functions
        - Listener function
        - Trigger function
    - Dissecting function-level nano architecture
        - Models
        - Connectors
        - Handlers
    - Choosing between REST and GraphQL
        - REST
        - GraphQL
    - Implementing different kinds of BFF services
        - CRUD BFF services
        - List-of-values (Lov) BFF services
        - Task BFF services

        - Search BFF services
        - Action BFF services
        - Dashboard BFF services
        - Reporting BFF services
        - Archive BFF services
    - Summary
7. Bridging Intersystem Gaps
    - Creating an anti-corruption layer
        - Macro-level ports and adapters
        - Substitution principle
        - External bulkhead
    - Dissecting the External Service Gateway pattern
        - Connectivity
        - Semantic transformation
            - Ingress
            - Egress
        - Packaging
        - Separate cloud accounts
    - Integrating with third-party systems
        - Egress – API call
        - Ingress – webhook
        - Asynchronous request response
        - Synchronous ESG/BFF hybrid
    - Integrating with other subsystems
        - Egress – upstream subsystem
        - Ingress – downstream subsystem
    - Integrating across cloud providers
    - Integrating with legacy systems
        - Ingress – Change Data Capture
        - Egress – Direct SQL
        - Egress – circuit breaker
        - Ingress – relay
    - Providing an open API and SPI
        - Ingress API – event
        - Ingress API – command
        - Egress SPI – webhook
        - Egress API – query

- Tackling common data challenges
  - Idempotence
  - Enriching data
  - Latching and cross-referencing
  - Slow data resync
- Summary

8. Reacting to Events with More Events
- Promoting inter-service collaboration
  - Choreography’s pros and cons
- Dissecting the Control Service pattern
  - collect
  - correlate
  - collate
  - evaluate
  - emit
  - expire
- Orchestrating business processes
  - Entry and exit events
  - Parallel execution
- Employing the Saga pattern
  - Compensating transactions
  - Abort events
- Calculating event-sourcing snapshots
  - What is ACID 2.0?
  - Snapshot events
- Implementing complex event processing (CEP) logic
  - Decision tables
  - Missing events
- Leveraging machine learning (ML) for control flow
  - Models
  - Predictions
- Summary

9. Running in Multiple Regions
- Justifying multi-regional deployment
  - Regional disruptions
  - Nominal cost
  - Higher user satisfaction

- Choosing a regional topology
  - Primary/hot-secondary
  - Active/active
    - Balanced users
    - Balanced data processing
- Preparing for regional failover
  - Offline-first
  - Protracted eventual consistency
- Checking regional health
  - Traditional health checks
  - Regional health checks
    - Metrics
    - Regional health check service
- Configuring regional routing
  - Content Delivery Network (CDN)
  - API Gateway
    - Regional endpoints
    - Global endpoints
  - CDN plus API Gateway
  - Global bus
- Replicating across regions
  - Change event vs domain event replication
  - Multi-master replication
  - Round-robin replication
- Dissecting regional failover
  - Query failover
  - Command failover
  - Trigger failure points
  - Listener failure points
- Addressing intersystem differences
  - External global endpoints
  - Legacy regional endpoints
- Implementing multi-regional cron jobs
  - Inserting job records and events
  - Relying on replication and idempotence
- Summary

10. Securing Autonomous Subsystems in Depth

- Shared responsibility model
- Securing cloud accounts
  - Accounts-as-code
  - Identity access management
- Securing CI/CD pipelines
  - Pipeline permissions
  - Deployment service permissions
  - Permission boundaries
- Securing the perimeter
  - Global/edge infrastructure
  - Encryption in transit
  - Federated identity
- Securing the frontend
  - OpenID Connect
  - Conditional rendering
  - Protected routes
  - Passing the JWT to BFF services
- Securing BFF services
  - JWT authorizer
  - JWT assertion
  - JWT filter
  - Last modified by
  - Least privilege
- Redacting sensitive data
  - Envelope encryption
  - Crypto-shredding
- Securing ESG services
  - Securing shared secrets
  - Using API keys
- Auditing continuously
  - Build-time and runtime auditing
  - Change event audit
- Summary

11. Choreographing Deployment and Delivery
    - Optimizing testing for continuous deployment
      - Continuous discovery
      - Continuous testing

- Focusing on risk mitigation
  - Small batch size
  - Decoupling deployment from release
  - Feature flags
    - Natural
    - Keystone
    - Artificial
  - Fail forward fast
- Achieving zero-downtime deployments
  - The Robustness principle
  - Between the frontend and its backend
  - Between producers and consumers
  - Between the backend and its data store
- Planning at multiple levels
  - Experiments
  - Story backlog
    - Story mapping
    - Story planning
  - Task roadmaps
- Turning the crank
  - Task branch workflow
    - Create task branch
    - Create draft pull request
    - Ready for review
    - Merge to master
    - Accept canary deployment
  - Feature flipping
    - Exploratory testing
    - Beta users
    - General availability
- Dissecting CI/CD pipelines
  - Continuous integration pipeline
    - Serverless testing honeycomb
    - Unit testing
    - Integration testing
    - Contract testing
    - Transitive end-to-end testing

        - Continuous deployment pipeline
            - Regional canary deployments
            - Synthetics
    - Summary
12. Optimizing Observability
    - Failing forward fast
    - Turning observability inside out
    - Leveraging FinOps
        - Cost is a metric
        - Dissecting monthly cloud cost
        - Worth-based development
    - Collecting resource metrics
        - The USE method
        - Capacity and concurrency limits
        - Throttling, iterator age, and queue depth
        - Errors and fault events
    - Tracking system events
    - Alerting on work metrics
        - The RED method
        - BFF service metrics
        - Domain event metrics
        - Anomaly detection
    - Observing real user activity
        - Real User Monitoring (RUM)
        - Synthetics
    - Tuning continuously
        - Function memory allocation
        - Cold starts
        - Timeouts and retries
        - Cache-control
    - Summary
13. Don’t Delay, Start Experimenting
    - Gaining trust and changing culture
        - Establishing a vision
        - Building momentum
        - Constructing an architectural runway
        - Seeding and splitting teams

- Funding products, not projects
  - Architecture-driven funding
  - Team capacity-driven funding
- Dissecting the Strangler pattern
  - Event-first migration
  - Micro frontend – headless mode
  - Retirement
- Addressing event-first concerns
  - System of record versus source of truth
  - Duplicate data is good
  - Avoid false reuse
- Poly everything
  - Polyglot programming
  - Polyglot persistence
  - Polycloud
- Summary

Other Books You May Enjoy

Index

Preface

Welcome to the book *Software Architecture Patterns for Serverless Systems*! Our industry has been searching for a silver bullet that allows us to quickly implement high-quality software. While there has been no single development that helps us do this, there has been the culmination of many developments that, when brought together in the right combination, help us to drive down lead time and continuously deliver business value.

Lean methods, DevOps practices, and the cloud have played a significant role in helping us to continuously deliver high-quality solutions. Now it is our software itself that has become the bottleneck and thus impedes our ability to deliver. So, my goal for this book is to show how to architect systems that enable change and system evolution. This is a comprehensive reference guide to designing architecture and implementing proven practices for delivering business value in a dynamic environment.

This book shows how the **serverless-first** mindset empowers **autonomous teams** and how the **event-first** approach creates an **inversion of responsibility** that allows us to create enterprise-scale serverless systems. You will learn how to apply the SOLID principles to define a system architecture with well-defined boundaries and then fortify these boundaries with autonomous services and autonomous subsystems. Add in micro frontends and finally, you'll be able to architect your own event-driven, serverless systems that are ready to adapt and change, so that you can deliver value at the pace needed by your business.

To get the most out of this book, be prepared with an open mind to discover why serverless is different. Serverless forces us to rewire, how we reason about systems. It tests all our preconceived notions of software architecture. I have personally found delivering serverless solutions to be, the most fun and satisfying. So, be prepared to have a lot of fun building serverless systems.

Who this book is for

This book is for software architects and aspiring software architects who want to learn about different patterns and best practices to design better software. Intermediate-level experience in software development and design is required. Beginner-level knowledge of the cloud will be beneficial.

What this book covers

Chapter 1, *Architecting for Innovation*, teaches architects that their job is to facilitate change and that their architecture must enable change. We survey the forces that impact lead time and look at the history of different integration styles to see how this influences the architectural decisions behind all the design patterns. Then we introduce autonomous services and the CPCQ flow and enumerate the set of concepts that enable teams to drive down lead times and continuously deliver business value.

Chapter 2, *Defining Boundaries and Letting Go*, is where architects begin to divide and conquer the problem domain. An architect will learn how to apply SOLID principles and event-first thinking to define architectural boundaries by dividing a system into autonomous subsystems and those subsystems into autonomous services. They will also learn the importance of continuous governance and how to rely on automation, observability, and key performance indicators to assert the performance of individual teams and to perpetuate an architectural feedback loop.

Chapter 3, *Taming the Presentation Tier*, addresses the challenges at the presentation tier and provides modern solutions, such as micro frontends, offline-first techniques, and more.

Chapter 4, *Trusting Facts and Eventual Consistency*, covers the foundational patterns that turn events into facts and support asynchronous inter-service communication, such as the event hub, systemwide event sourcing, and event stream processing.

Chapter 5, *Turning the Cloud into the Database*, covers the core data patterns that fight data gravity by turning the database inside out and

spreading it out along the phases of the data life cycle, including **Command Query Responsibility Segregation** (**CQRS**) and **Change Data Capture** (**CDC**).

Chapter 6, A Best Friend for the Frontend, covers the Backend for Frontend pattern that supports end-user activities and micro applications throughout the data life cycle.

Chapter 7, Bridging Intersystem Gaps, covers the External Service Gateway pattern that supports interactions with external systems and creates an anti-corruption layer that protects the system from everything outside the system, including third-party systems, legacy systems, other autonomous subsystems, and more.

Chapter 8, Reacting to Events with More Events, covers the Control Service pattern that supports business process orchestration, sagas, complex event processing, machine learning, and more.

Chapter 9, Running in Multiple Regions, describes how to run serverless systems in multiple regions and prepare for regional failover with regional health checks, latency-based routing, data replication and more.

Chapter 10, Securing Autonomous Subsystems in Depth, introduces the shared responsibility model for serverless computing and covers topics such as securing CI/CD pipelines, data reaction and more.

Chapter 11, Choreographing Deployment and Delivery, describes how to decouple deployment from release so that teams can continuously deploy and deliver with zero downtime. It covers the core concepts of backward compatibility and quality assurance.

Chapter 12, Optimizing Observability, describes how to turn serverless metrics into actionable information so that teams can experiment with

confidences and fail forward fast.

Chapter 13, Don't Delay, Start Experimenting, provides thoughts on how to start making forward progress toward the architectural vision provided throughout the book. It addresses some of the myths and anti-patterns that keep teams from getting off the ground and covers various hurdles and impediments that architects must confront.

To get the most out of this book

To follow along and experiment with the templates provided with this book, you will need to configure your development environment according to the following steps:

1. Install Node Version Manager (https://github.com/nvm-sh/nvm) or (https://github.com/coreybutler/nvm-windows)
2. Install the latest version of Node.js with `nvm install node`
3. Create an AWS account (https://aws.amazon.com/free) and configure your credentials for the Serverless Framework (https://www.serverless.com/framework/docs/providers/aws/guide/credentials)
4. Download the templates: (https://github.com/jgilbert01/templates) and (https://github.com/jgilbert01/micro-frontend-template)
5. Run the project scripts, such as:

```
$ npm ci
$ npm test
$ npm run test:int
$ npm run dp:np:w
```

Download the example code files

The code bundle for the book is hosted on GitHub at https://github.com/jgilbert01/templates and https://github.com/jgilbert01/micro-frontend-template. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: https://packt.link/gbp/9781803235448

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “We can implement the same role-based assertions in GraphQL by decorating the model with directives, such as `@hasRole`e.”

A block of code is set as follows:

```
export const forRole = (role) => (req, res, next) => {
  const groups = getUserGroups(req);
  if (groups.includes(role)) return next();
  else res.error(401, 'Unauthorized');
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
 toUpdateRequest: (uow) => ({
    Key: {
      pk: uow.event.thing.id,
      sk: 'Thing',
    },
    ...updateExpression({
      name: uow.event.thing.name,
      ttl: uow.event.timestamp + (60*60*24*33) // 33 days
    }),
  }),
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “**Software as a Service** (**SaaS**) products typically provide a well-documented open API for invoking their functionality.”

Warnings or important notes appear like this.

Tips and tricks appear like this.

There is a concise diagramming convention used throughout this book. Serverless services have multiple resources, which can clutter diagrams with a lot of connecting arrows. The following sample diagram demonstrates how we minimize the number of arrows by placing related resources adjacent to each other so that they appear to touch. The nearest arrow implies the flow of execution or data. In this sample diagram, the arrow on the left indicates that the flow moves through the API gateway to a function and into the database, while the arrow on the right indicates the flow of data out of the database stream to a function and into the event bus. These diagrams are created using Cloudcraft (https://cloudcraft.co).

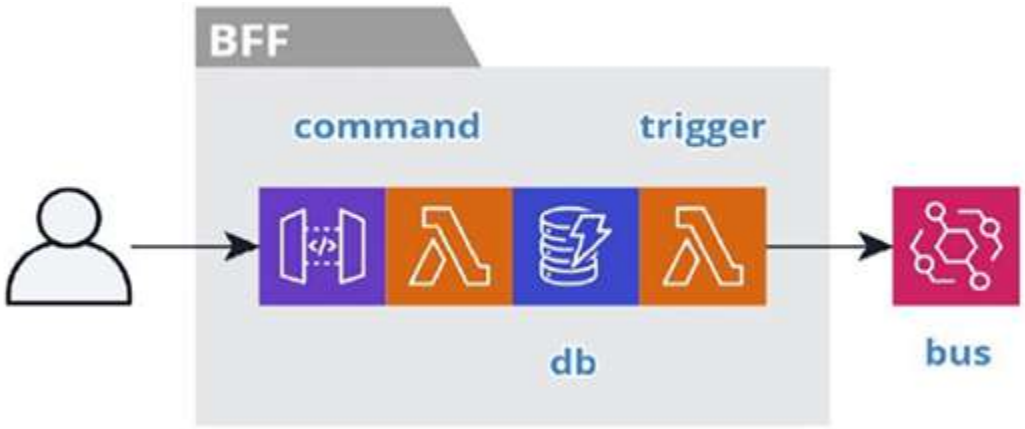


Figure: Demonstration of flow

The following table provides a legend of the most common icons used in the diagrams throughout the book.

Icon	Resource	AWS	Azure	GCP
	Event Bus	Event Bridge	Event Grid	Eventarc
	Event Stream	Kinesis	Event Hubs	Pub/Sub
	Function	Lambda	Functions	Functions
	Datastore	Dyanmo DB	Cosmos DB	Firestore
	Object Store	S3	Blob Storage	Cloud Storage
	API Gateway	API Gateway	API Mgt	API Gateway

Figure: Resource icon legend

The software architecture patterns in this book are cloud provider agnostic. However, I use AWS to implement the examples and AWS-specific icons in the diagrams. The preceding table enumerates various cloud provider resources that you could use to implement these patterns on the major cloud providers.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share your thoughts

Once you've read *Software Architecture Patterns for Serverless Systems, Second Edition*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don’t worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don’t stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/9781803235448

2. Submit your proof of purchase
3. That’s it! We’ll send your free PDF and other benefits to your email directly

1

Architecting for Innovation

Change. Change is the only constant in the software industry. There are countless reasons for this change. All too often, the needs for our software change faster than we can write and we cannot adapt fast enough.

This is our quest in this book and in our industry. We need to architect our software for *change*. We must *innovate* at the speed of business. We need to adapt and change our software in real time. We need to *experiment* to find the best solutions in a dynamic environment. And we need the *confidence* to move fast and innovate, without breaking what works.

To achieve this, we first need to know what holds us back. We need to understand our software development past so that we improve going forward. So, in this first chapter, we will start with an analogy of many software projects and then consider an analogy that we can aspire to. Then we will survey our software development practices and integration styles to see how they increase *lead time* and slow us down. Finally, we will take this knowledge and sketch out the serverless event-driven architecture that we will unfold throughout the book: an architecture that gives *autonomous* teams the confidence to continuously *innovate* and deliver *business value*.

In this chapter, we're going to cover the following main topics:

- Continuously delivering business value
- Dissecting lead time
- Dissecting integration styles
- Enabling autonomous teams with autonomous services

Continuously delivering business value

Why do we write software? It's a simple—but important—question. The answer may seem obvious, yet, as with so many things, the answer is much more nuanced than it seems at first.

For me, the answer depends on which role I am playing. Am I playing the role of developer or architect? As a developer, I expect that my answer is the same as that of most developers. My short answer is: "Because I can, because it is fun, and it's what I do." I can create things, seemingly out of thin air. The creative process itself is its own reward. I enjoy what I do!

I have spent most of my career building custom software solutions. When someone who is not in the industry asks me what I do, I like to use an analogy; the typical custom motorcycle shop reality TV show serves as a great analogy for most software development projects.

By the skin of our teeth

Most of us have seen this type of show, and they all follow a similar formula. The customer wants a cool, custom motorcycle but only has a vague idea of what the final product should be. The shop owner is a wizard at turning the customer's idea into an amazing design that they cannot resist. They agree on a price and a totally unrealistic schedule. Now, it is up to the team to turn the design into reality, no matter what it takes.

They do it on a compressed schedule and a shoestring budget because they enjoy creating things and seeing their customers love what they create. Never mind the fact that the process was filled with enough drama to keep the audience entertained. Mistakes were made, workarounds were devised,

the requirements changed, rework was needed, and the team barely held it together along the way. Working around the clock, the team delivers, and the customer loves the results.

The problem with this analogy is that it hits a little too close to home. This really sums up the kind of experience many—if not most—software development teams live through. But the ends don't justify the means. This is *unsustainable*.

Through high-velocity teamwork

As a **chief technical officer** (**CTO**) and chief architect, my goal is to do better than delivering by the skin of our teeth. We build software to help achieve the vision and mission of the organizations we work for. Continuously delivering business value must be our primary focus. Yet, we need to create *sustainable* practices and an architecture that supports them.

To this end, I much prefer the analogy of a Grand Prix motorcycle racing team. Manufacturers assemble these teams with the core purpose of driving *innovation* and outperforming the competition. They take the current product and push it to its limits and beyond. They collect copious *metrics* and scrutinize rider *feedback*. They formulate new experiments, rebuild the bike to the new specs, and run the tests again, pushing the limits ever farther. It is an amazing example of *high-velocity teamwork and continuous improvement*.

These teams achieve an inspiring level of sustainability as they continue to produce value, lap after lap, race after race, and season after season. There is drama, but it comes from their desire to outdo the competition by delivering the best-performing product.

This is an example of what we are striving for with our craft. We write software to relentlessly and *continuously deliver business value*. As software development teams, we accept that we work in a *dynamic environment* of fickle customers and stiff competition. We expect our business strategies to evolve and shift. Our methodologies and practices have become much more accommodating of constant change, and the cloud enables us to react more quickly.

However, we need more than just techniques and technology to succeed and flourish in this environment. We must *innovate* to stay ahead and deliver *value*. In other words, we must *experiment* with different *changes* to our software. **Our architecture must enable change**—this is our focus in this book. Our *methodology* and our *architecture* have to work *together* to support the sustainable pace of innovation that we need to achieve. The bottom line is this: we must consciously **architect for innovation**.

Let's start by looking at the forces that impact our ability to deliver timely solutions.

Dissecting lead time

Our ability to continuously deliver business value is a function of *lead time*. If we measure our lead times in months or even weeks, then our ability to deliver is not continuous. We are wasting valuable time while we assume that we are building the right solution. Much like the Grand Prix motorcycle analogy, we want to put our software in users' hands as fast as possible so that they can test drive it and provide *feedback*. Then, we tune the solution and quickly send it back out for another test drive.

To become a high-velocity team, we need to build the *muscle memory* that allows us to produce this seemingly uninterrupted flow of ideas and

experiments. However, we need to get out of our own way. The typical software product delivery pipeline is full of potential *bottlenecks* that increase lead time. We need to understand these bottlenecks before we can produce an architecture that enables change.

Let's survey the forces that influence our lead time. We will start at the beginning of the pipeline and work our way through to see what optimizations we need so that we can take control of our lead times.

Risk mitigation

Risk is a powerful motivator. Traditionally, we mitigated risk by slowing down and increasing lead time. We believed this allowed us to double- and triple-check our results before releasing software to users. Yet, in today's dynamic environment, the biggest risk is not delivering the right solution to users or not delivering it in time to be effective. More often than not, the solution we set out to deliver is not the solution we land on, yet when we slow down, it takes that much longer to learn that we are off the mark. Then, it is all the more likely that we will not deliver a timely solution.

Instead, we need to frame our original idea as a hypothesis that needs validation. Then, we put the software in the hands of real users as quickly as possible so that we can find out if we are heading in the right direction and adjust the course before it is too late. Driving down lead times and moving at this velocity allows us to respond quickly to user feedback and iterate to the right solution. Along the way, we are actually delivering more and more value.

However, going faster introduces the risk of breaking things that already work. Users rely on the existing features and expect them to work. Users

always want more, but not at the expense of losing what they already have. So, we need to *move fast without breaking things*.

The fear of breaking things naturally causes teams to slow down, so we need an architecture that gives us confidence to move fast. Our architecture must mitigate the risk of breaking things by limiting the blast radius of any mistakes to just what is changing.

But we need buy-in from decision-makers.

Decision making

Decision-making and a willingness to make decisions is another factor that influences lead times. Traditionally, companies manage software products as projects. To move forward, a project must have a plan, a budget, and approval. The process of achieving this is time-consuming and expensive, which produces an incentive to create fewer but larger and more expensive projects. This, in turn, creates a vicious cycle that further increases lead time. Ultimately, the risk of approving such a large budget limits the effectiveness of the project-based approach.

Fortunately, companies are beginning to understand that these traditional methods do not work in today's dynamic environment. They are learning that it is more effective to define the desired outcomes and then give teams the autonomy to make smaller, targeted decisions and work toward the desired outcomes. As architects, we need to promote this change by providing leadership and guiding the business to more effective management practices for software products. We will cover this topic in *Chapter 13, Don't Delay, Start Experimenting*.

In the meantime, we will put a methodology and an architecture in place that enables us to react quickly as the business shifts priorities.

Software development life cycle methodology

The **Software Development Life Cycle** (**SDLC**) methodology a team follows is the most obvious factor that impacts lead time. **Waterfall** is the classic methodology and the king of long lead times, with software releases measured in *months to years*. Its most significant false assumption is that we can define all the requirements upfront. However, in a dynamic environment, this is not possible. We do not have the luxury of time. We must *discover* the requirements as we go.

Agile methods are a major improvement, with lead times measured in *weeks*. However, a 2-week Agile sprint, as an example, only produces production-ready software. It does not necessarily put the software into the hands of the actual end users in production. It is not uncommon for several sprints to pass before we actually release (that is, deliver) the software to real users in production. The net result is that we still have real lead times measured in *months*, which is too long.

Kanban and **Lean** methods do the best job at helping us control *batch size* so that we can put working software in the hands of real beta users with lead times measured in *hours to days*. However, to move at this pace, we need to learn how to mitigate the risk to working software by *decoupling deployments from releases*, and we need to build the muscle memory that allows us to work at this pace. The following diagram depicts this practice,

and we will cover this in detail in *Chapter 11*, *Choreographing Deployment and Delivery*:

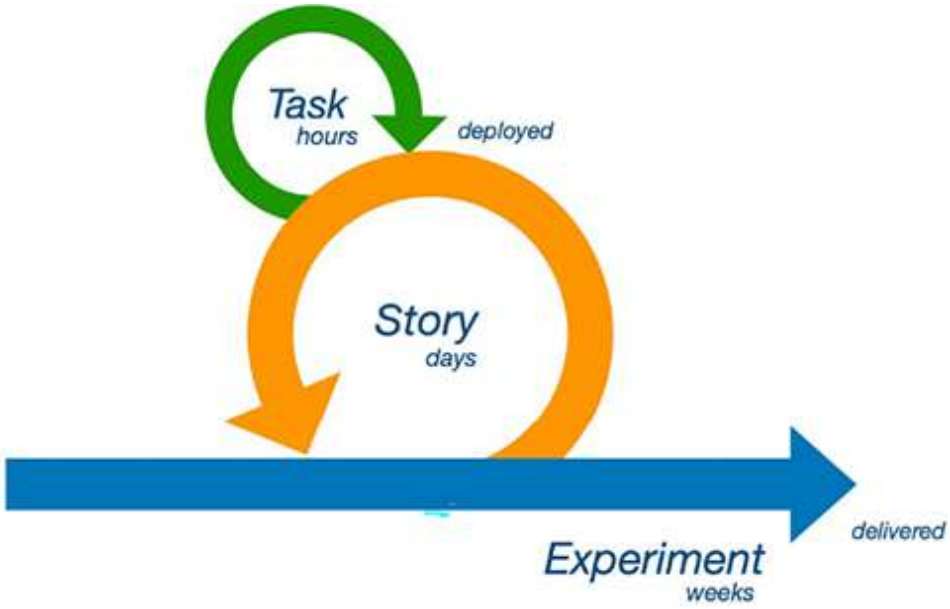


Figure 1.1: Decoupling deployment from release

But these practices require an architecture that enables change and allows us to work at this pace. So, let's continue down the product delivery pipeline to uncover more bottlenecks that impact lead time and influence our new architecture.

Hardware provisioning

Hardware provisioning is the cornerstone of lead time. Historically, we purchased these hard assets (that is, computers) as a capital expense. This required accounting, budgeting, and approval processes with long lead times. Then, we needed to purchase the hardware and have it delivered. After that, we needed to schedule resources to install and configure the hardware. The lead time for this process could easily take many months.

With hardware as the long pole in the tent, there was little incentive to optimize other parts of the software product delivery pipeline. The Waterfall method was well suited for these conditions, and Agile processes worked well if we provisioned hardware proactively. There was no reason to go faster.

Then, the cloud changed everything. We could provision hardware *on demand*. Hardware became an operating expense and no longer required long approval processes. The context changed, and this set a *domino effect* in motion that triggered the optimization of all downstream activities in the software product delivery pipeline. The next domino to fall was *manual* deployment.

Deployments

Before the cloud, there was little reason to *automate* hardware provisioning and software deployments. In my experience, deploying a new release of software was an error-prone, manual process that could easily take days, with all hands on deck to resolve seemingly endless configuration issues. At this pace, there was no incentive to deploy more often than every 2 to 3 weeks or so—it would simply be a waste of effort to deploy more frequently. But none of this mattered when the lead time for hardware was longer.

But the cloud turned *manual* deployments into the new long pole in the tent since hardware was now available on demand. We had an incentive to optimize this step in the delivery pipeline. It became imperative to automate the deployment of the entire stack—not just the software, but the infrastructure as well. We started treating the entire stack as **infrastructure-as-code** (**IaC**).

Deployment lead times plummeted. The quality and predictability of deployments increased dramatically. A deployment was no longer a major event—it was just routine. We began to develop the muscle memory that allowed us to deploy at will, but we ran headlong into the next bottleneck. Our software could not get out of its own way. We needed to change how we structured and architected software.

Software structure

How we deploy software has a natural impact on how we structure and architect software. Deploying software *manually* is a tedious process that creates an incentive to do fewer deployments. As a result, we gravitated toward a monolithic software architecture, which is completely understandable. Combining many features into a single unit of deployment eased the burden of performing deployments. However, as we began to automate software deployments, it became abundantly clear that combining features into monoliths was the next *bottleneck* in the software product delivery pipeline.

Our software itself was impeding our ability to change it and reduce lead times. We could iterate on the functionality and automate the deployments, but we were afraid to actually pull the deployment trigger for fear of the *unintended consequences* of redeploying the whole monolith for a small change in just one piece of the monolith. We could not control the *blast radius* if something went wrong. A mistake could bring down the entire monolith. So, we would wait until we could batch up multiple changes, which forced lead times back up and brought us full circle.

We needed to break up the monolith. *Automated deployments* made it practical to deploy many *microservices*. We could make small changes to

individual services and redeploy them independently. We could finally drive down lead times for individual services, and the lead times of each service would not impede each other—or so we thought. As we will see in the *Dissecting integration styles* section, dividing the monolith into microservices is not without its own side effects. But we were on the right track.

Yet prevailing questions remained. How would we slice and dice the monolith into smaller deployable units? How would we retain the development *simplicity* and *productivity* of the monolith? For example, how could a developer work on one service without having to run all the services? How would we maintain the *understandability* of the system as a whole? How would we avoid the inevitable *big ball of mud* that would become known as the Microlith? Worst yet, how would we avoid the dreaded microservices death star? Most importantly, how would we control the *blast radius* when we broke a microservice?

We will start to unravel these questions in the *Enabling autonomous teams with autonomous services* section. Then, we will begin our deep dive into the patterns and building blocks of our architecture in *Chapter 2*, *Defining Boundaries and Letting Go*. But first, let's continue our survey of the factors that impact lead time. Testing is the next bottleneck in our software product delivery pipeline.

Testing and confidence

Traditional testing methods have a mixed relationship with lead time and a rich history of marginalization. As projects approached their deadlines, time would simply run out, and testing teams had to leverage what limited time remained. Yet to go faster and drive down lead times, we actually need *more*

testing. We need to test continuously to mitigate the risk of breaking working software. This gives teams the confidence to go faster because, without confidence, teams will naturally put on the brakes and drive up lead time.

The problem is that our traditional testing methods are predicated on the same false assumption that we can define all the requirements upfront. Those methods no longer make sense because they require the software to proceed through a set of gates that drive up lead time when we need to knock it down.

Instead, our architecture will allow us to experiment. We will increase testability and observability. We will automate *continuous testing* and shift it to the left in the delivery pipeline to ensure there are no regressions in working software. We will leverage *feature flags* to control access to the new software. Then, we will put the new software in the hands of beta users so that they can validate our hypothesis. We will see how all these pieces fit together in *Chapter 11, Choreographing Deployment and Delivery*.

But to err is human. We will make mistakes, so we need to limit the *blast radius*. To do this, we need to understand the dependencies between the software that is changing and the working software, and this requires *inter-team communication*, which has a negative impact on lead time.

Dependencies and inter-team communication

Dependencies and inter-team communication have a significant impact on lead time. When one team is dependent on another, we cannot compress lead times any more than the time it takes for other teams to respond to our requests. When teams share resources, one team cannot make a change

without the consent of the others. We need to allocate additional time and effort to coordinate and align team schedules and priorities. Plus, teams will naturally put on the brakes when they are not confident that they will receive prompt assistance when they are in a pinch.

Conway's Law states that *organizations are constrained to produce designs that are copies of their communication structure*. In other words, to drive down lead times, our team structure and the structure of our software architecture need to work together. Teams cannot be at the mercy of other teams. They cannot share resources. They cannot rely on matrixed teams for needed skill sets. They must be completely *self-sufficient*. They must own the full stack. They must be *cross-functional*. Teams must be **autonomous**, and our software architecture must enable autonomous teams. In the *Serverless-first – creating knowledge* section, we will see how **self-service** cloud capabilities make this practical.

However, we cannot eliminate all dependencies in a system. A software system is a collection of many features, and those features are related to each other in natural ways. To maximize team autonomy and take control of lead times, we need an approach for creating autonomous features. We need an *inter-feature communication mechanism* that helps us mitigate risks and gives teams the confidence to experiment.

So let's look at the different *integration styles* that allow features to communicate. In this section, we optimized our software delivery pipeline by removing the bottlenecks. In the next section, we will see how our choice of integration style influences our architecture and our lead times.

Dissecting integration styles

Our software systems are composed of many features that must work together to deliver value for the end users. How these features communicate has been one of the most significant topics in our industry. The communication pathways within a system impact the timely flow of information, but they also impact our ability to deliver timely solutions. A highly integrated system will maximize the flow of information. However, if we integrate features too tightly, then it impedes our ability to adapt to changing requirements. It can even put the stability of a system at risk as we make changes.

Our architecture must balance these opposing concerns. Let’s survey the different integration styles to see how they impact our systems, so we can take the best parts from each and bring them together in our new architecture.

Batch integration

When I started my career in software development, the major pain point for the industry was information silos. Companies had multiple systems throughout their organizations, and they were not well integrated. Valuable business information was isolated in these silos (that is, systems). Data in one system was not shared with the others. Duplicate manual data entry was the order of the day.

The integrations that did exist were batch-oriented. At best, these jobs ran on a nightly schedule. This meant that the data was often stale and even in conflict. The reconciliation of data between the silos was often a nightmare. The dependencies between the silos were an afterthought. The flow of information between them was too slow, which diminished the value of the overall system.

But the silos were *autonomous*. They could change on their own schedule, and downtime in one did not cause downtime in another. The silos were eventually consistent. It was just a matter of when, and at what cost.

Spaghetti integration

To make matters worse, the batch integrations were *point-to-point*. Each job only took a single source system and a single target system into consideration. This resulted in what we commonly refer to as *spaghetti integration*. The following diagram depicts a worst-case scenario where every system (that is, node) integrates with every other system in both directions. This results in *N (N-1) / 2* integrations, where *N* = number of nodes. In reality, this depiction is much cleaner and more predictable than what we see in practice:

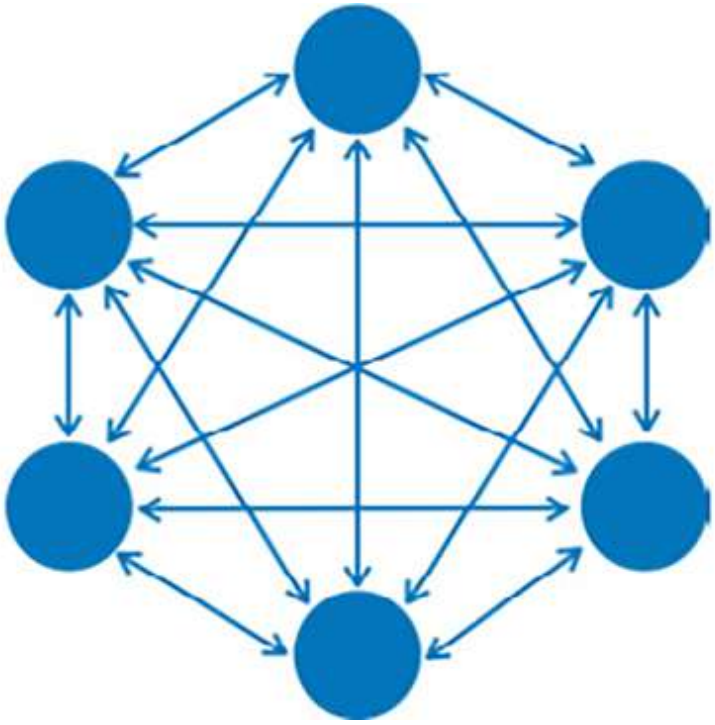


Figure 1.2: Point-to-point integration style

This approach creates a maintenance nightmare. A change to one system can result in weeks—or even months—of rework (that is, lead time) to upgrade all the integrations. Procrastination is a common way to avoid this problem. Meanwhile, the lead time for needed improvements grows and grows as teams *resist change*. Without a clean integration approach, the dependencies between the systems are a barrier to change.

Real-time integration

The industry was looking for a new integration approach, and the client-server model seemed like the solution. We started writing real-time integrations between the new **graphical user interface** (**GUI**) applications and the legacy systems. From the end user perspective, this seemed like a good thing because the latest information was immediately available. However, we had actually made things worse.

We had made everything *tightly coupled*. The *availability* of one system was now dependent on the availability of the systems it integrated with. A mistake in one system could now ripple through and incapacitate many systems. We had eliminated the reconciliation problem by shifting the burden to the end users. The system would remain consistent, but the users could not accomplish their work.

We had undervalued the *autonomy* that the information silos provided. We optimized one part of the equation, instead of *optimizing the whole*. Information was available in real time when everything was running smoothly, but it wasn't available at all when things went wrong.

The net result was an increase in lead time. The new real-time integration style was still point-to-point spaghetti integration. We needed more inter-

team communication and coordination to keep those *brittle* integrations working.

Enterprise application integration

Along the way, a new approach to integration emerged. The deregulation of the 1990s resulted in many acquisitions and mergers. Companies found themselves with duplicate systems. Consolidating these systems was a lengthy process, so we needed to integrate them in the meantime. This led to the emergence of the **enterprise application integration** (**EAI**) industry and paradigm.

The duplicate systems needed to be integrated in such a way that they did not need to change. Each would still be the master of its own data, so we needed to support multiple heterogeneous masters. We implemented bi-directional synchronization between all the systems. Eventually, all but one of the duplicate systems would be retired by just turning them off once all users were transitioned.

The resulting approach was a near-real-time, eventually consistent, event-based synchronization with a *hub-and-spoke* model that supported *plug-and-play*. The following diagram depicts this optimized solution where each application (that is, node) integrates in both directions with a *canonical model*. This reduced the maintenance burden to $N - 1$ integrations, where N = number of nodes:

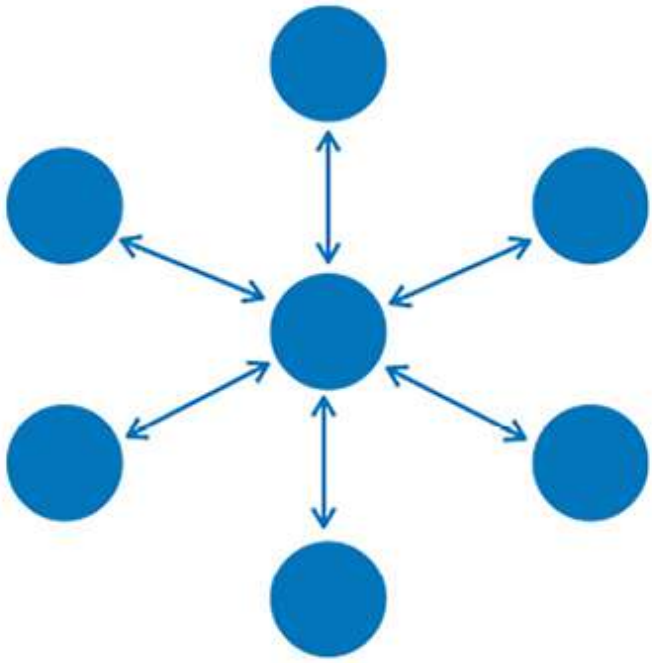


Figure 1.3: Hub-and-spoke integration style

This approach was a major improvement. The systems remained decoupled and autonomous, with no single point of failure. The low latency of near-real time significantly lessened the need for reconciliation to account for errors in eventual consistency. The technical debt and complexity of point-to-point spaghetti integration were eliminated, and lead time was kept in check since teams did not have to coordinate with every other team.

Yet this approach did not reach widespread adoption.

Shared database

Instead, the trend shifted toward the consolidation of data within a mammoth monolithic shared database. The relational database became the center of the system. All applications interacted with the same data. An update from one application was immediately visible in other applications. The development model was simple and straightforward.

But the database was now the single point of failure. Any downtime for the database meant downtime for the entire system. A change to the shared schema could require changes to many applications. A change for one application required approval by a committee to help ensure that it did not break other applications. The net result was an increase in lead time.

Service-oriented architecture

Meanwhile, **service-oriented architecture** (**SOA**) captured the industry's imagination, and **enterprise service bus** (**ESB**) middleware flooded the market. SOA added an additional abstraction layer between the applications and the database, which eased the burden of data model changes and limited the impact of a database schema change in the service layer. This was a move in the right direction.

However, the bloated middleware increased lead time and decreased performance. The ESB was another shared resource that we needed to procure, provision, configure, and manage. As with all shared resources, the central team that owned the ESB was overworked, so lead times increased, and performance suffered because every request had to compete for ESB resources.

Microservices

Eventually, the deficiencies of SOA, combined with the impedance mismatch of monolithic software deployments, led to the adoption of microservices. We were finally starting to move back in the direction of autonomous systems. Each microservice was independently deployable and had its own database. Microservices did not share resources. A single team

owned a microservice and all its resources. This helped keep a lid on the need for inter-team communication.

However, we did not change our programming model and integration style. We took all our best practices for the monolith and superimposed them on microservices, creating **microliths** instead. Despite all its flaws, real-time synchronous integration was still the preferred approach, but the volume of microservices and their interdependencies magnified these flaws many times, as illustrated in the following image:

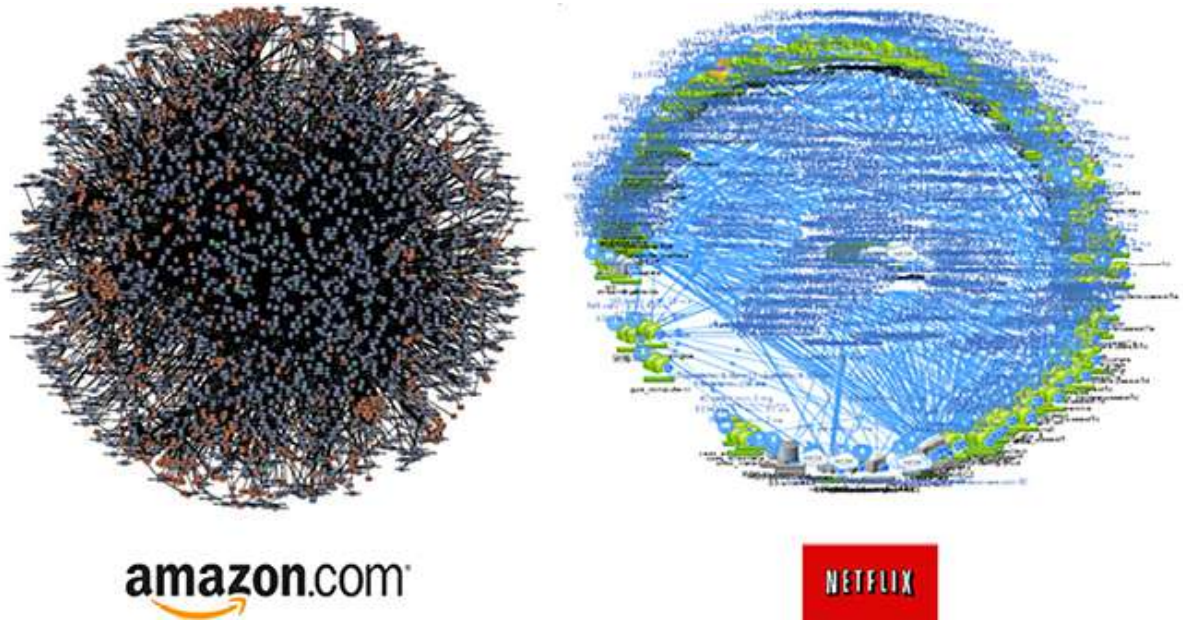


Figure 1.4: Microservices death star

Figure 1.4 depicts examples of the inevitable **microservices death star** that is created when we leave all these dependencies unchecked. It is the modern equivalent of spaghetti integration. The individual microservices themselves become the shared resources. An outage in a highly dependent microservice would ripple through the system, and a change to such a microservice could require reworks across many microservices.

Lead time did improve for small, self-contained changes, but a mistake in any change still represented a risk to the system. Lead time for any significant change remained high. We still needed extensive inter-team coordination to keep these brittle integrations working. All in all, the complexity of these systems skyrocketed.

To drive down lead times and continuously deliver business value, our architecture needs to bring together the best of all these integration styles. We need the **autonomy of information silos** and the **simplicity of the database programming model**. We need the **level of abstraction provided by SOA** and the **efficiency and loose coupling of EAI**, and we need the **independence of microservices without the complexity**. We need to balance the need for timely information with the need for timely solutions. We need to *optimize the whole*.

Now we are ready to establish our architectural vision and cover the core concepts of the architecture that enables change.

Enabling autonomous teams with autonomous services

As we have just seen, our ability to drive down lead times and iterate to innovative solutions is impeded by many forces. Lean methods and the cloud help us eliminate waste, but our software itself is an impediment to innovation. The complex dependencies within our systems necessitate inter-team communication and coordination, which consumes valuable time. The high coupling between components creates the risk of cascading failures when things go wrong. So, teams naturally slow down for fear of breaking things.

We need a new architectural vision. Our architecture must enable autonomous teams to move fast and limit the blast radius of honest mistakes. The following diagram depicts this architectural vision. Imagine a topology of autonomous services that we design for integration from the ground up:

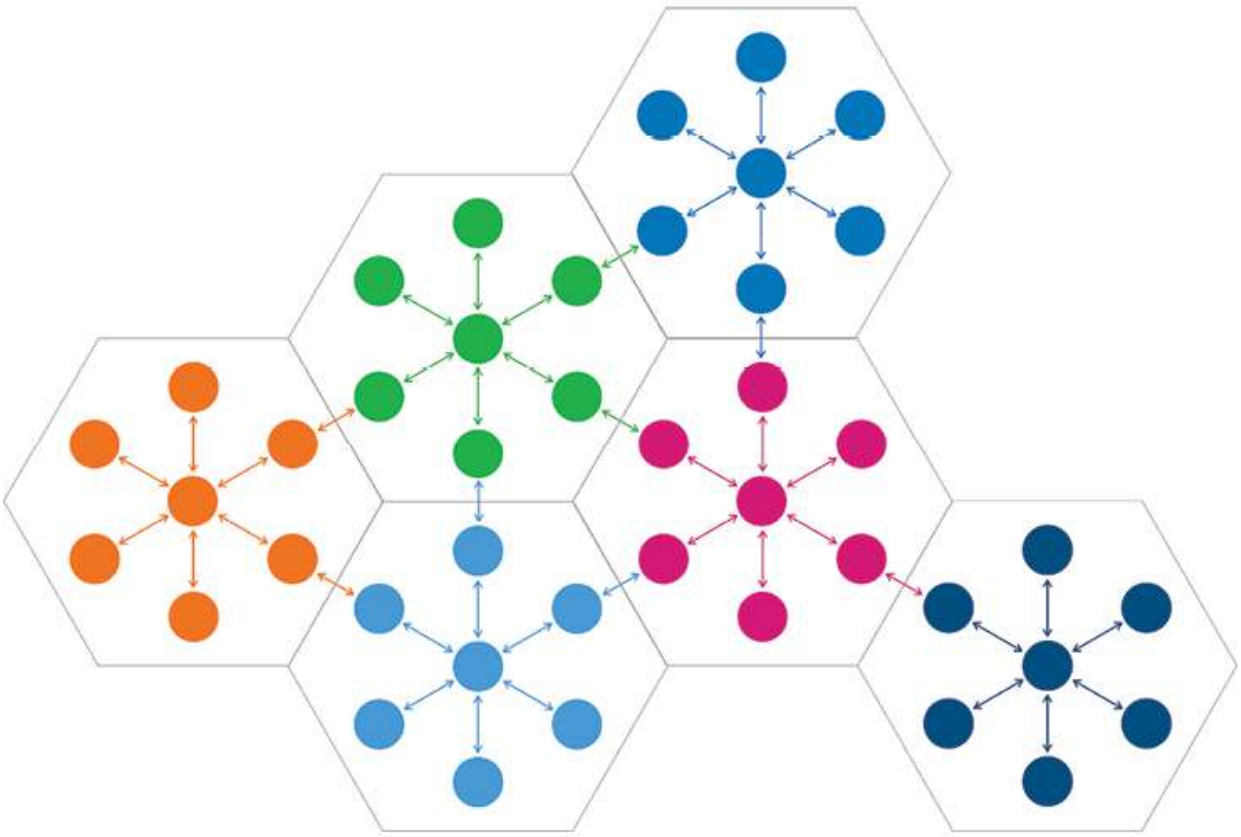


Figure 1.5: Event-first topology of autonomous services and subsystems

Each autonomous service owns all the resources it needs to continue to function even when other services cannot. We group autonomous services into autonomous subsystems. At the heart of each subsystem is an event hub that eliminates direct inter-service communication. Following an event-first approach, these loosely coupled services exchange events through their event hub. Then, we connect autonomous subsystems together to create more complex systems, by simply exchanging pertinent events.

In *Chapter 2, Defining Boundaries and Letting Go*, we will see how to decompose a system into autonomous subsystems and those subsystems into autonomous services. Now, let's dive deeper into autonomous services and cover the core concepts of this architectural vision.

Autonomous services – creating bulkheads

To go fast and *not* break things, we need resilient services. We need services that continue to function when related upstream and downstream services cannot. We want teams to be confident that an honest human error on their part will not bring down the whole system.

At the same time, we need to keep the overall complexity of the system in check. Simple is always better. We want to compose our systems from simple and repeatable building blocks that we snap together. We want the system to naturally evolve by simply adding and removing uncoupled services.

Each autonomous service will fall under one of our three higher-order autonomous service patterns: **Backend for Frontend** (**BFF**), **External Service Gateway** (**ESG**), or **Control Service**. Each of these patterns is straightforward, easily recognizable, repeatable, and autonomous. With these key patterns, we can create highly complex systems composed of many of these easily understandable services.

Developers can easily work on one autonomous service without needing to pull in others. They can easily reason about the internals of the service they are changing. Most importantly, they can easily understand the impact of a given change on the well-defined external interaction points.

We accomplish all this by simply creating bulkheads between the services. These bulkheads allow services to *buffer outgoing changes* and *cache incoming data*. The following diagram depicts the basic building blocks that give services their autonomy and allow them to communicate. I call these building blocks the **Command, Publish, Consume, Query** (**CPCQ**) flow:

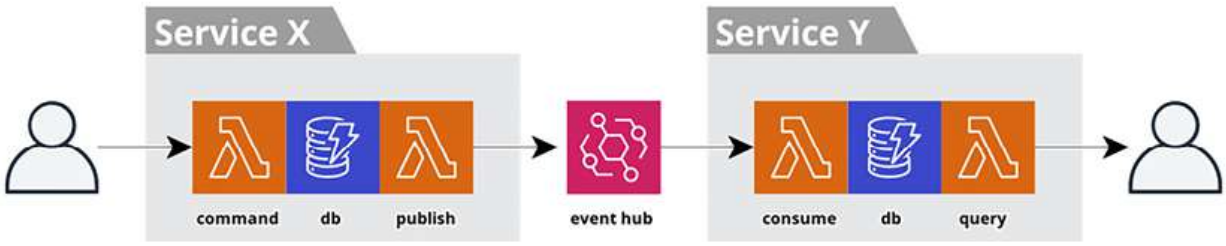


Figure 1.6: CPCQ flow

The **CPCQ** flow divides inter-service interactions into a series of *atomic actions*. These atomic actions create bulkheads between the services that eliminate the need for inter-service *synchronous* communication. We can *chain* these flows together to create ever-evolving systems.

The primary patterns that enable this **CPCQ** flow are the *event hub and system-wide event sourcing* patterns, which we cover in *Chapter 4*, *Trusting Facts and Eventual Consistency*, along with *system-wide CQRS*, which we cover in *Chapter 5*, *Turning the Cloud into the Database*.

Let’s see how the **CPCQ** flow creates service autonomy.

Asynchronous inter-service communication

In the *Dissecting integration styles* section, we saw how the *synchronous* real-time integration style increases the coupling between the components of a system. While it does have a positive impact on the timely delivery of

information to the user, it also has a negative impact on our ability to deliver timely solutions, due to the increased need for inter-team communication and coordination. So, we need to strike a balanced approach that optimizes the system as a whole.

To give services their autonomy, we need to place restrictions on the use of synchronous communication. We will restrict all *synchronous* communication to *intra-service* interactions, and we will use *asynchronous* communication for all *inter-service* interactions. This will ensure the responsiveness of the **user experience** (**UX**) and maintain the simplicity of the **developer experience** (**DX**) while eliminating the coupling to other services.

As depicted in *Figure 1.6*, the **CPCQ** flow clearly identifies these boundaries. Each service maintains its own data store. We will perform synchronous *commands* and *queries* against these data stores. Upstream services will asynchronously *publish* events to the event hub as their internal state changes. Downstream services will asynchronously *consume* events and cache lean data in their own data store.

Fortified boundaries

Synchronous inter-service communication also increases the risk of widespread outages, since a failure in one service can have a ripple effect through all dependent services. This risk increases lead time because teams will naturally put on the brakes to avoid it. To mitigate this risk, we must put bulkheads in place.

A bulkhead is a construct placed between two components that acts as a protective barrier to prevent the spread of a problem from one component to another. On the one hand, architecture is the *art* of defining boundaries

within a system. On the other hand, architecture is the *science* of placing bulkheads along these boundaries. We can summarize this by saying **the architecture of a system is defined by its fortified boundaries**.

As depicted in *Figure 1.6*, the **CPCQ** flow clearly identifies these bulkheads. The bulkheads are the highly available, fully managed, serverless data stores we place throughout the system to *buffer* and *cache* data as it moves through the system. These include both *outbound* and *inbound* bulkheads.

The **event hub** provides the *outbound bulkhead* that protects upstream services from downstream services. A service can *publish* events to the event hub without knowledge of the downstream consumers or their availability. The event hub will *persist* and *buffer* the events until downstream services can *consume* them.

The **data store** of an individual autonomous service provides the *inbound bulkhead* that protects a downstream service from upstream services. A downstream service *consumes* events from the hub and creates a lean replicated *cache* in its own data store. The downstream service is no longer dependent on the availability of upstream services. It maintains access to the last known values of the data it needs to function. This data store also acts as an intermediate *buffer* for any changes the service makes to its data, before publishing to the event hub.

With synchronous communication under control and bulkheads in place, we can contain the blast radius of honest mistakes. Teams are free to innovate at their own pace. They can confidently reduce their lead times to days if not hours. When we make an honest mistake, upstream and downstream services will continue to operate unabated. The broken service will recover quickly,

as the team relies on its *muscle memory* to fail forward fast, and downstream services will become consistent shortly thereafter.

Some coupling remains since downstream services are dependent on the structure of the domain events produced by upstream services. But we can keep this coupling under control by practicing the *robustness principle*, as we will cover in *Chapter 11*, *Choreographing Deployment and Delivery*. Plus, we will add an abstraction layer in the form of *external domain events*, which we will cover in *Chapter 2*, *Defining Boundaries and Letting Go*, and we will use the ESG pattern to create an *anti-corruption layer* in *Chapter 7*, *Bridging Intersystem Gaps*.

Now, let's see how an *event-first* mindset makes the system more flexible, as well as more resilient.

Event-first – valuing facts

Achieving an event-driven enterprise has been the holy grail of software architecture for as long as I can remember. However, **event-driven architecture** (**EDA**) has struggled to reach widespread adoption and rise to its potential. Event-based programming is indispensable at the GUI tier, and the event-driven model of **Function-as-a-Service** (**FaaS**) is a giant leap in the right direction. Yet enterprises continue to create fragile microservice death stars because they believe they are less complex.

So we need to take a fresh look at EDA and see how it is different this time around. Our autonomous services build on the experience of previous EDA generations. A new *event-first* mindset helps us look at these systems in the right light and build systems that are more flexible and resilient.

Inversion of responsibility

One concern with EDA revolves around the impression that these systems are more complex and difficult to debug and manage. There is definitely a paradigm shift. However, our autonomous services and the **CPCQ** flow give us the best of both worlds. On the front side of these services, we have the traditional *synchronous* actions that make for a simple and straightforward development experience. On the back side, we have the *asynchronous* actions that produce events and make our systems more flexible.

By focusing first on the events that our services produce, we create an *inversion of responsibility* that pushes key design decisions downstream. We design each service to support a specific actor. *Upstream* services publish events to record the fact that an actor performed an action, without concern for what happens next. *Downstream* services take on the *responsibility* of *reacting* to these events as they see fit.

This focus on the *verbs* of the business domain, instead of its *nouns*, naturally decouples the features of a system and makes it more flexible and adaptive. It enables change and innovation through *organic evolution*. We simply add or remove services to change how the system *reacts*. This will become evident as we start applying the *SOLID* principles to our architecture in *Chapter 2, Defining Boundaries and Letting Go*.

Events as first-class citizens

Most importantly, we will be treating events as *first-class* citizens this time around. We have traditionally treated events as *ephemeral* messages—in other words, we deleted the events after we processed them. The only record we had that these events ever existed was the side effects they had on their consumers. This meant that when things went wrong, we had little recourse other than to restore the consumer's state from the source systems. These systems were not self-correcting. They required manual intervention. They

required data reconciliation. *This, by and large, was the reason for EDA's bad reputation*.

This time around, events represent the system *contract*, and we place a high value on events as the *facts* of the system. We store the events in an *event lake* in perpetuity. This store of events is a *record of truth* for a system. It provides a complete *audit log* of the history of a system and is a valuable source of *metrics*. When things go wrong, we can *replay* these events to repair broken services. We can also replay the events to seed a new service or to investigate a hypothesis. We will dig deeper into this shift in thinking in *Chapter 2, Defining Boundaries and Letting Go*, and *Chapter 4, Trusting Facts and Eventual Consistency*.

Idempotence and ordered tolerance

In previous EDA generations, things often went wrong because we placed our faith in the false assumptions that we could guarantee the orderly arrival of messages. This time around, we will be taking a realistic and practical stance on the value of *exactly-once* and *ordered delivery* of events. We prefer these characteristics because they improve the throughput of the system when things are behaving normally. However, when things do go wrong—as they inevitably will—then these unrealistic expectations result in processing and data discrepancies. Instead, our autonomous services are *idempotent* and *order-tolerant*. We will cover the patterns and techniques that help us make our services accommodating and self-correcting in *Chapter 4, Trusting Facts and Eventual Consistency*, *Chapter 5, Turning the Cloud into the Database*, and *Chapter 8, Reacting to Events with More Events*.

Asynchronous messaging infrastructure is also notoriously difficult to manage. Resource limitations were often the root cause of problems, and a lack of observability made it difficult to intervene proactively. Let's see how

a *serverless-first* approach helps address these issues, while also empowering autonomous teams.

Serverless-first – creating knowledge

One of the hard lessons I learned from my experience in the early days of lifting, twisting, and shifting on-premises monoliths to the cloud is that managing resources—such as servers, databases, and queues—in the cloud is extremely difficult. Autoscaling was sold as the be-all and end-all but while scaling up was easy, scaling down was hard, and it is the edge case nuances that can bring a system to its knees when new instances cannot start up faster than the current instances are crashing down.

These are often self-inflicted scenarios that we could never dream of, such as subtle memory leaks and deadlocks. You have to experience them to believe them. It can easily take a team many months to develop the skills and experience to safely run workloads in production. Meanwhile, lead time is increasing, as well as the risks of delivering the wrong solution.

Self-service

To make matters worse, it became clear that our reliance on traditional relational databases and messaging systems would not be sufficient at a cloud scale. There were many new alternatives to choose from, and it was reasonable to expect that we would need multiple options to support our various features.

This meant that it would not be practical for each autonomous team to acquire all of the necessary skills. This would push us back to having central

teams managing these resources or—even worse—actually sharing the resources. Teams would not be in control of their lead times. Waiting on other teams would impede their ability to innovate.

This was about the time that serverless emerged. Serverless is *self-service*. We could offload the central teams to the cloud provider. We did not need to share resources. Teams could provision resources at will. These resources were highly available and highly observable. Teams could get up and running with new technology quickly. We could safely go into production without the learning curve.

Disposable architecture

Yet, we weren't creating vendor lock-in. These decisions were *disposable*. Teams could start with *serverless-first* and change later if it made sense. As an example, when I wanted to experiment with streaming technology for the first time, I decided to apply two *Lean* principles: *defer commitment* and *create knowledge*. Instead of trying to manage my own *Kafka* infrastructure, I decided to use the fully managed, cloud-native alternative, **Amazon Web Services** (**AWS**) Kinesis. This was new ground. We didn't know what we didn't know. This would allow us to experiment and quickly learn the general concepts, and even go into production without all the risk and complexity of running the technology ourselves. But we didn't commit to this choice. Over time, we could switch if we outgrew the managed solution.

This was an example of *serverless-first* and *disposable architecture* at work. Using a serverless alternative requires little upfront investment. And these decisions are not monolithic—we can make and unmake these decisions on a *service-by-service* basis. This makes these architectural decisions completely *disposable*. Conversely, when teams put a lot of time and effort into learning

to wield specific technologies, we tend to become emotionally attached to the solutions. It's human nature. As a result, we tend to double down on our commitment to these architectural decisions, even when they are the wrong solution.

Instead, we should start with a serverless solution first and create knowledge. We learn about the technology. But more importantly, we learn, as fast as possible, whether or not we are on the right track to business value. Maybe some features will ultimately need self-managed technology, but that is difficult to predict upfront and would impede innovation. *Deferring commitment* is just the cost of creating this knowledge.

Once a team knows that their feature does in fact provide business value, they can then leverage the observability and cost metrics of their serverless technology to make an informed decision about whether or not it is worth switching. In the meantime, they can leverage serverless-first as a *safety net*. But, in my experience, autonomous services and the CPCQ flow of atomic actions, are a perfect match for serverless, and moving them off will likely increase cost and risk.

The bottom line is that *serverless-first* **makes autonomous teams practical**. Teams are in control of their own lead times. They don't have to depend on other teams. They have the peace of mind that the technology is stable, scalable, secure, disposable, and under their control. They can focus their time and effort on iterating to an optimal solution and delivering business value.

Now, let's see how *data gravity* impacts our lead times and our architecture.

Data life cycle – fighting data gravity

Back in 2003, I was working on an archiving strategy for a large enterprise accounting system. The system had a relational database with hundreds of tables and years and years of data, and all our customers expressed concern over managing the size of the database. No matter where I started pulling on the thread of the relational data model, everything seemed to be related to everything else through some cycle in the relationships. The database simply wouldn't let me delete a significant amount of data, even though I knew it would be OK. In essence, the pull of the data's gravity had everything locked in place.

I began thinking about the relationship between our **online transaction processing** (**OLTP**) database and our **online analytics processing** (**OLAP**) database. It was our OLTP database that was intractable. The OLAP database contained all the same data, but the nightly **extract, transform, load** (**ETL**) processes translated and duplicated the data into many different formats to suit the needs of the different analytics users. Archiving the OLAP database was not an issue. It wasn't an explicit requirement, but we had accidentally designed the OLAP database to be archivable. Conversely, the OLTP data had been optimized and sub-optimized, and tortured and tormented, so to speak, so that a single copy of the data could suit the needs of many masters. The OLTP data violated the **single-responsibility principle** (**SRP**) because it had more than one reason to change; it supported more than one actor.

I came to realize that different user groups used the same logical data throughout its life cycle, at different times, and for different purposes. Each phase had its own storage and access requirements. Each had its own archiving requirements and could keep its data as lean as possible. Some

could simply purge older data after a **time to live** (**TTL**), while others needed a restore capability. There were generally four *phases* in the data life cycle, as depicted in the following image:

Figure 1.7: Data life cycle

We already had an EAI backbone for integrating different applications. So, why not break up the OLTP database and the related functionality into separate applications along the data life cycle, with separate databases, and even different database technology, and synchronize the data in near-real time?

Focusing on the life cycle of our data is a fundamental change in how we design and architect systems. Our object-oriented roots taught us to zero in on the *nouns* of our data models and encapsulate that data with all its behaviors. This has the negative impact of coupling all the requirements of the data life cycle phases together, which increases *data gravity* and reduces agility, as any change to the database schema must go through a committee.

With our autonomous services and event-first approach, we will focus on the verbs instead. We will leverage the BFF pattern and tailor services to specific actors to perform specific functions (that is, verbs) throughout the data life cycle. These services will encapsulate their own lean versions of the logical data model and optimize the data at each stage in its life. The system-wide event sourcing and CQRS patterns will provide the mechanisms to move the data throughout its life.

The *serverless-first* approach makes this *polyglot persistence* practical for all teams. We will turn our traditional databases inside out, spreading the data

across the capacity of the cloud and, ultimately, turn the cloud into the database. We will eliminate data gravity and liberate data so that our architecture enables change instead of crushing all innovation under the weight of the data. We will dive into the details of this technique in *Chapter 5, Turning the Cloud into the Database*, and in *Chapter 6, A Best Friend for the Frontend*.

Micro frontends – equalizing tiers

Innovations at the presentation tier are never-ending. The number of technical alternatives for implementing the UX is constantly churning. One innovation in particular is the **single-page application** (**SPA**) written in JavaScript. This approach has created many opportunities to reduce lead time. For example, we can simply deploy an SPA bundle to the edge of the cloud and serve it to the browser where the client device executes the presentation logic. We no longer have to run any servers for the presentation tier—it is completely serverless. This is a great win for autonomous, self-sufficient, full-stack teams.

But there's still more to be done at the presentation tier. We need to break up the monolith of the presentation tier. Autonomous services are a big win for reducing lead time because they allow us to deploy features independently. However, when we continue to create monolithic frontend applications, we marginalize this improvement. To reap the full benefits of autonomous teams, we need to create an autonomous stack from top to bottom. To maximize their throughput, autonomous teams need equal and full control of all the tiers: presentation, service, and data.

Micro frontends are another important innovation. This approach breaks the presentation tier into a set of independently deployable micro applications but stitches them back together for a seamless UX. The combination of the micro frontends, BFF, and CQRS patterns provides autonomous teams with the freedom to experiment and iterate to the optimal business value. In *Chapter 3, Taming the Presentation Tier*, we will see how we are finally beginning to control the churn at this tier.

Observability – optimizing everything

Observability is a critical concept in **quality assurance** (**QA**) that helps give teams the confidence to increase their rate of deployment. As with the motorcycle racing team analogy, the systems we build must provide a window into their inner workings. Armed with these metrics, we can observe a system at runtime and assert its health. A system essentially emits signals that we can test for anomalies. We can alert the team on the **key performance indicators** (**KPIs**) so that they can jump into action when a feature fails. Simply knowing that we are actively collecting and monitoring this information bolsters team confidence. Instead of fearing deployments, teams can focus on improving their **mean time to recovery** (**MTTR**) by driving down lead times and creating *muscle memory*.

Observability enables continuous governance. We can monitor cloud resource configuration changes for compliance, particularly with regard to security compliance. This enables development organizations to scale because architects and security experts are no longer a bottleneck to innovation. Autonomous teams are free to innovate, with these configuration monitors acting as a *safety net*. Metrics on team performance, such as lead

time and MTTR, also provide insight into the maturity of the teams. All these observability metrics help architects focus their efforts on teams that need the most help.

We can also use metrics about user traffic and feature utilization to gauge the success or failure of our experiments—for example, an experiment may hypothesize that a specific change will drive some additional percentage of traffic to a specific resource. Observability is necessary to evaluate the hypothesis.

Observability also enables worth-based development. Teams can leverage cost and performance metrics to help decide how to best apply their limited team capacity. For example, we can continuously tune the performance of services based on real runtime data, up to the point where the additional development effort is more expensive than the cost savings generated.

The bottom line is that we need to measure and observe anything and everything we want to optimize. We will cover observability topics in *Chapter 2, Defining Boundaries and Letting Go*, in *Chapter 10, Securing Autonomous Subsystems in Depth*, and in *Chapter 12*, *Optimizing Observability*.

Organic evolution – embracing change

As architects, we must feel comfortable working in a dynamic environment of continuous change. This is an environment where the real risk is not being able to change course fast enough as circumstances and priorities change. We must be at ease with spearheading an architecture that enables change.

Yet, as engineers, we tend to prefer a concrete plan. We like to be able to connect all the dots before committing to a course of action.

But this is no longer realistic. Maybe it never was. We have to be willing to embrace change, both in the ways we think about architecture and in the methods we use to deliver valuable business solutions. We optimized the architecture of our legacy systems to suit the era of long lead times, but disruptive changes in technology have changed all the rules. We are now able to take full control over our lead times, and we need to architect our systems accordingly.

Our job is to harness this creative disruption and drive change throughout the organization. We need to make these changes so that we can deliver timely business value. This includes changes in architecture, methodologies, and practices, and ultimately, in culture and bureaucracy.

These changes will not happen overnight. They will organically evolve the same way we expect our software to organically evolve: by performing experiments. Autonomous teams love this new freedom. They immediately see how these changes positively impact their daily lives. The harder changes lie upstream in the software product delivery pipeline, and we must take them head-on. These include changes to practices, such as budgeting and governance.

We have just touched on the core concepts in this chapter, and we will dive into the details in the remaining chapters. In the last chapter, *Chapter 13, Don't Delay, Start Experimenting*, we will cover some of the anti-patterns and hang-ups that impede progress. Continuous change and organic evolution will feel much more manageable once we have rewired our mental picture and internalized how all these pieces fit together. But I can guarantee,

based on my own experience, that you will have more fun *architecting for innovation*.

Summary

In this chapter, you learned that the role of architecture is to enable change so that autonomous teams can continuously deliver business value. You learned about the various forces that impede a team's ability to drive down lead times and iterate to an optimal solution, and you learned how dependencies and integration styles impact architecture.

We dug into the details of autonomous services, and you learned how to fortify the boundaries between services so that autonomous teams can have the confidence to move fast without breaking things. Along the way, you learned why the event-first approach treats events as facts, how a serverless-first strategy creates knowledge and makes autonomous teams practical, and why data gravity impedes progress.

In the next chapter, we will begin to define architectural boundaries by dividing a system into autonomous subsystems and those subsystems into autonomous services. Then, we will address the importance of continuous governance.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/B8TubSxc35

2

Defining Boundaries and Letting Go

In *Chapter 1, Architecting for Innovation*, we learned that the role of architecture is to enable change so that autonomous teams can confidently and continuously deliver business value. The key concept here is autonomy, but we need more than just autonomous teams. We need an architecture that promotes autonomy. We ultimately accomplish this by creating autonomous services with fortified boundaries.

But before we can fortify our boundaries, we need to **define** those boundaries. This is where our *functional architecture* emerges and dovetails with our *technical architecture*. We will define our boundaries at *multiple levels*, including the **subsystem**, **service**, and **function** levels. We define these boundaries up front so that autonomous teams can work in parallel with guard rails in place as they experiment within their boundaries.

In this chapter, you will learn how to define these *boundaries* and we will introduce our *autonomous service patterns*. Then we will dig into the details in the remaining chapters. But first, we will review the proven concepts and guiding principles that help us shape our boundaries so that they are meaningful and flexible.

At the tail end of the chapter, we will touch on the hardest thing of all for architects to do, which is to let go and trust the autonomous teams to deliver within their boundaries. Here, we will look at how we can govern without impeding innovation.

In this chapter, we’re going to cover the following main topics:

- Learning the hard way
- Building on proven concepts
- Thinking about events first
- Dividing a system into autonomous subsystems
- Creating subsystem bulkheads
- Dissecting an autonomous subsystem
- Dissecting an autonomous service
- Governing without impeding

Learning the hard way

You may be wondering why it is so important to define architectural boundaries. We all want to jump right in and start coding. But it is easy to get lost in the details of a new project, and we can find ourselves on a slippery slope if we do not set our bearings from the start.

I had a run-in with architecture early in my career that left an indelible impression on me. It was the 90s and n-tiered architecture was all the rage. I was the architect of a system with a middle tier that we wrote in C++ and ran on Tuxedo. This was well before **Continuous Integration** (**CI**) had emerged, so we lived by our nightly builds. One morning I arrived at work and found that the nightly build was still running. It was a large system with many subsystems, multiple services, and a significant quantity of code, but a nightly build that took all night was an indication that something was wrong. The system was still growing, so things would only get worse if we did not identify and remediate the root cause.

It didn’t take long to find the root cause, but the solution, although simple, would be tedious to roll out. In C++, we define classes in header files and

include these files where we use the classes. However, there is no restriction on how many classes you can define in a header file. Our domain entities encapsulated our private persistence classes, but all these private classes were leaking, because we defined them in the same header files. We were building this private code over and over again, everywhere that we used the domain entities. As the system grew, the cost of this mistake became more and more expensive.

The SOLID principles did not exist at that time, but in essence, the system violated the **Interface Segregation Principle**. The header files contained more interfaces than necessary. The simple solution was to move these internal classes into private header files. We immediately began to implement all new features this way and a few strategic updates brought the builds back under control.

But the project was behind schedule, so the rest of the changes would have to wait. I took it upon myself to make those changes as time permitted. It took six months to retrofit the remainder of the system. This experience **taught me the hard way** about the importance of clean architectural boundaries.

More often than not, the hard way is the best way to learn. We need to *discover* the right solutions for our end users. We need to get our hands dirty and *experiment* to find out what works and what does not. But we need to define architectural boundaries at *multiple levels* that will act as guard rails as we perform our experiments.

As an industry, we have already learned a lot about what makes for good clean architecture. So, let’s look at some of the proven concepts we will be building on.

Building on proven concepts

If you are new to serverless computing, then you may be wondering how serverless architecture differs from more traditional architectures. Of course, there are differences when we get down to the details. But, by and large, serverless architecture builds on the same proven concepts that we should employ no matter what software we are writing.

We will use **Domain-Driven Design** (**DDD**), apply the SOLID principles, and strive to implement a clean Hexagonal Architecture. One of the more interesting differences, though, is that we will apply these concepts at multiple levels. We will apply them differently at the **subsystem level** (that is, *macro architecture*), the **service level** (that is, *micro architecture*), and the **function level** (that is, *nano architecture*).

Let's review these concepts and see where we will apply them in our serverless architecture and how we will do so differently than we have in the past.

Domain-driven design

When I started my career, our industry was struggling with the paradigm shift to object-oriented programming. The **Unified Modeling Language** (**UML**) eventually emerged, but we all used it in different ways. The Gang of Four's book *Design Patterns*, Gregor Hohpe's and Bobby Woolf's book *Enterprise Integration Patterns*, and Martin Fowler's book *Patterns of Enterprise Application Architecture* helped us create better designs. But it wasn't until Eric Evans' book *Domain-Driven Design* took root that we all had common semantics that we could use to produce more familiar designs.

Let’s review the main concepts of DDD that we will employ throughout this book: bounded context, domain aggregate, and domain event.

Bounded context

A **bounded context** defines a clear boundary between different areas of a domain. Within a bounded context, the domain model should be consistent and clear. Between bounded contexts, we use **context maps** to define the relationships between the models. Defining these *boundaries* is the main thrust of this chapter. We will be using this concept to define our autonomous subsystems and therefore our cloud accounts as well. In *Chapter 7, Bridging Intersystem Gaps*, we will learn how to create an **anti-corruption layer** to implement the *context mapping* between subsystems (that is, bounded contexts).

Domain aggregate

A **domain aggregate** is a top-level domain entity that groups related entities together in a cohesive unit. We will exchange data between services and subsystems at this aggregate level. We will cover data modeling and introduce the idea of *data life cycle* architecture in *Chapter 5, Turning the Cloud into the Database*. We will introduce the **Backend For Frontend** (**BFF**) pattern later in this chapter and in *Chapter 6, A Best Friend for the Frontend*, we will see how each BFF service owns a domain *aggregate* at a specific phase in the *data life cycle*.

Domain event

A **domain event** represents a change in state of a domain aggregate. These events are fundamental in our event-driven architecture and the heart of our

autonomous services. We will cover the idea of event-first thinking in this chapter and *event processing* in *Chapter 4, Trusting Facts and Eventual Consistency*.

And of course, we will strive to use a **ubiquitous language** within each bounded context.

SOLID principles

SOLID is an acronym for a set of guiding design principles that help us implement clean, flexible, and maintainable software. The principles include the following:

- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Robert Martin first promoted these principles for improving object-oriented designs. Since then, these principles have proven equally valuable at the *architectural* level. Although the implementation of the principles varies at different levels, it is convenient to apply a consistent mental framework throughout. Let’s see how we can use each of these principles to guide the creation of evolutionary architectures that enable change.

Single Responsibility Principle

The **Single Responsibility Principle** (**SRP**) states that a module should be responsible to one, and only one, actor.

The SRP is simultaneously the most referenced and arguably the most misunderstood of all the SOLID principles. At face value, the name of this principle suggests *a module should do one thing and one thing only*. This misunderstanding leads to the creation of fleets of microservices that are very small, very interdependent, and coupled. Ultimately, this results in the creation of the *microliths* and *microservice death stars* that I mentioned in the first chapter. The author of this principle, *Robert (Uncle Bob) Martin*, has made an effort to correct this misunderstanding in his book *Clean Architecture*.

The original and common definition of this principle states *a module should have one, and only one, reason to change*. The operative word here is **change**. This is crucial in the context of architecture because the purpose of architecture is to enable change. However, when we apply this principle incorrectly, it tends to have the opposite effect, because highly interconnected modules impede change. The coupling increases the need for inter-team communication and coordination and thus slows down innovation.

In his latest definition of the SRP, Uncle Bob is focusing on the source of change. **Actors** (that is, people), drive changing requirements. These are the people that use the software, the stakeholders, and the owners of external systems that interact with the system, even the governing bodies that impose regulations. After all, people are the reason that software systems exist in the first place.

The goal of this principle is to avoid creating a situation where we have **competing demands** on a software module that will eventually tie it in knots and ultimately impede change. We achieve this goal by defining architectural boundaries for the different actors. This helps ensure that the

individual modules can change independently with the whims of their actors. Uncle Bob also refers to this as the **Axis of Change**, where the modules on either side of the boundary change at different rates.

Later in this chapter, we will use the SRP to divide our software system into autonomous subsystems and to decompose subsystems into autonomous services. The architectural boundaries will also extend vertically across the tiers, so that the presentation, business logic, and data tiers can all change together.

Open-Closed Principle

The **Open-Closed Principle** (**OCP**) states that *a module should be open for extension but closed for modification*. *Bertrand Meyer* is the original author of this principle.

This principle highlights how human nature impacts a team's pace of innovation. We naturally slow down when we *modify* an existing piece of software, because we have to account for all the impacts of that *change*. If we are worried about unintended side effects, then we may even resist change altogether. Conversely, adding new capabilities to a system is far less daunting. We are naturally more confident and willing to move forward when we know that the existing usage scenarios have been completely untouched.

An architecture that enables change is open to adding and experimenting with new features while leaving existing functionality completely intact to serve the current user base. With the SRP, we define the architectural boundaries along the axis of change to limit the scope of a given change to a single service. We must also close off other services to any inadvertent side effects by fortifying the boundaries with *bulkheads* on both sides.

Events will be our main mechanism for achieving this freedom. We can extend existing services to produce new event types without side effects. We can add new services that produce existing event types without impacting existing consumers. And we can add new consumers without changes to existing producers. At the presentation tier, **micro frontends** will be our primary mechanism for extension, which we will cover in *Chapter 3, Taming the Presentation Tier*.

Sooner or later, modification and refactoring of existing code is inevitable. When this is necessary, we will employ the **Robustness** principle to mitigate the risks up and down the dependency chains. We will cover common scenarios for extending and modifying systems with zero downtime in *Chapter 11, Choreographing Deployment and Delivery*.

Liskov Substitution Principle

The **Liskov Substitution Principle** (**LSP**) states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program. Barbara Liskov is the original author of this principle, hence the L in SOLID.

The substitution principle is essential to creating evolutionary architecture. Most innovations will consist of incremental changes. Yet, some will require significant changes and necessitate running multiple versions of a capability simultaneously. Following the LSP, we can substitute in new versions, so long as they fulfill the contracts with upstream and downstream dependencies.

The LSP will play a major role in the hexagonal architecture we will cover shortly. We will use **events** to define the contracts for inter-service communication. This **design-by-contract** approach enables the substitution

that powers the **branch-by-abstraction** approach. We can substitute event producers and swap event consumers. We will leverage the LSP to strangle legacy applications as we modernize our systems and continue this evolutionary process indefinitely to support continuous innovation with zero downtime. But zero downtime requires an understanding of all the interdependencies, and this leads us to the Interface Segregation Principle.

Interface Segregation Principle

The **Interface Segregation Principle** (**ISP**) states that no client should be forced to depend on interfaces they do not use.

I provided an anecdote at the beginning of this chapter that highlighted the build-time issues that can arise when we violate the ISP. These build-time issues can have a big impact on a monolith. However, they are of less concern for our autonomous services because they are independently deployable units with their own CI/CD pipelines. We are still concerned about including unnecessary libraries because they can have an impact on cold start times and increase the risk of security vulnerabilities. But our real concern is on the deployment and runtime implications of violating the ISP and how it impacts downtime.

Our goal is to create an architecture that enables change so that we can reduce our lead times, deploy more often, and tighten the feedback loop. This requires confidence that our deployments will not break the system. Our confidence comes from an understanding of the scope and impact of any given change and our certainty that we have accounted for all the side effects and minimized the potential for unintended consequences. We facilitate this by creating clean and lean interfaces that we segregate from other interfaces to avoid polluting them with unnecessary concerns. This minimizes coupling

and limits scope so that we can easily identify the impacts of a change and coordinate a set of zero-downtime deployments.

A common mistake that violates the ISP is the creation of general-purpose interfaces. The misperception is that reusing a single interface will accelerate development time. This may be true in the short term, but not in the long term. The increased coupling ultimately impedes innovation because of competing demands and the risk of unintended consequences.

This is a primary driver behind creating client-specific interfaces using the **Backend for Frontend** pattern. We will cover this pattern in detail in *Chapter 6, A Best Friend for the Frontend*.

For all our inter-service communication, our individual domain events are already nicely segregated, because we can change each of them independently. We do have to account for the fact that many downstream services will depend on these events. We will manage this by dividing the system into subsystems of related services and using **internal domain events** for intra-subsystem communication and **external domain events** for inter-subsystem communication. From here, we will leverage the **Robustness** principle to incrementally evolve our domain events. We will see this in play in *Chapter 11, Choreographing Deployment and Delivery*.

Even with well-segregated interfaces, we still need to avoid leaky abstractions. This occurs when details of specific upstream services are visible in the event payload, and we inadvertently use those details in services downstream. This leads us to the Dependency Inversion Principle.

Dependency Inversion Principle

The **Dependency Inversion Principle** (**DIP**) states that a module should depend on abstractions, not concretions.

At the code level, we also refer to the DIP as programming to an interface, not to an implementation. We also refer to it as **Inversion of Control** (**IoC**). It manifests itself in the concept of **Dependency Injection** (**DI**), which became an absolute necessity in monolithic systems. It eliminates cyclic dependencies, enables testing with mocks, and permits code to change and evolve by substituting implementations while holding interfaces closed to modification. We will use simple constructor-based DI in our serverless functions without the need for a heavy-weight framework.

At the architecture level, I think we can best understand the value of the DIP by using the scientific method as an analogy. Holding variables constant is a crucial component of the scientific method. When we perform a scientific experiment, we cannot derive anything from the results if nothing was held constant because it is impossible to determine cause and effect. In other words, some level of stability is a prerequisite for the advancement of knowledge. **Stability** in the face of change is the whole motivation behind autonomous services. We need the ability to continuously change and evolve a running system while maintaining stability.

The DIP is a fundamental principle for creating architecture that provides for flexibility and evolution while maintaining stability. For any given change to a service, we have to hold something constant to maintain the stability of the system. That constant in an event-first system is the **domain events**. When we modify any service, all others will rely on the stability of the event types they all share. This will control the scope and impact of the change so that teams have the confidence to move forward.

Translating the DIP to the architectural level, we get the following:

- Domain events are the abstractions and autonomous services are the concretions.

- Upstream services should not depend on downstream services and vice versa. Both should depend on domain events.
- Domain events should not depend on services. Services should only depend on domain events.

In fact, many upstream services won't know or care that a downstream service exists at all. Downstream services will depend on the presence of an upstream service because something has to produce a needed event, but they should not know or care which specific upstream service produced an event.

This ultimately means that upstream services are responsible for the flow of control and downstream services are responsible for the flow of dependencies. In other words, upstream services control when an event is produced and downstream services control who consumes those events. Hence, the name of the principle still applies at the architectural level. We are inverting the flow of dependency (that is, consumption) from the flow of control (that is, production).

This event-based collaboration will be most evident when we cover the *Control Service Pattern* in *Chapter 8, Reacting to Events with More Events*. These high-level services embody control flow policies and other services simply react to the events they produce.

Taking the DIP to the next level is the notion that downstream services react to domain events, and upstream services do not invoke downstream services. This is an **inversion of responsibility** that leads to more evolutionary systems. It allows us to build systems in virtually any order and gives us the ability to create end-to-end test suites that don't require the entire system to be running or even completely implemented. This stems from the power of *Event-First Thinking*, which we will cover shortly. But first, let's review the concepts of Hexagonal Architecture.

Hexagonal Architecture

Building on the SOLID principles, we need an architecture that allows us to **assemble** our software in *flexible* ways. Our software needs the ability to execute in different runtime environments, such as a serverless *function*, a serverless *container*, or a *testing* tool. We may decide to change a *dependency*, such as switching from one type of datastore to another, or even switching cloud providers. Most importantly, we need the ability to *test* our domain logic in isolation from any *remote resources*. To support this flexibility, we need a **loosely coupled** software architecture that hides these *technical* design decisions from the domain logic.

Alistair Cockburn's *Hexagonal Architecture* is a popular approach for building loosely coupled modules. Robert Martin's *Clean Architecture* and Jeffrey Palermo's *Onion Architecture* are variations on this theme. An alternate name for this architecture is **Ports and Adapters**, because we connect modules through well-defined *ports* (that is, *interfaces*) and glue them together with *adapter* code. When we need to make a change, we leverage the SOLID principles, such as the DIP and LSP, and *substitute* different adapters.

Hexagonal architecture gets its name from the hexagonal shapes we use in the diagrams. We use hexagons simply because they allow us to represent systems as a *honeycomb* structure of loosely coupled modules. We read these diagrams from left to right. We refer to the left side as the **primary** or **driving** side because it defines the module's input flow, while we call the right side the **secondary** or **driven** side since it defines the output flow. The **inner** hexagon represents the clean domain **model** with its input and output **ports** and the **outer** hexagon holds the inbound and outbound **adapters**.

Hexagonal architecture emerged well before serverless technology and the popularity of the cloud. Its origins align with the development of more *monolithic* software systems. Today, our serverless solutions consistent of *fine-grained* resources that we compose into *distributed*, *event-driven* systems. As a result, the *shape* of our systems has changed, but their *nature* remains the same, and therefore, hexagonal architecture is more important than ever.

We need to scale hexagonal architecture up to a *macro* perspective and down to a *nano* perspective. In our *serverless* architecture we will apply the hexagonal concepts at three different levels, the **function**, **service**, and **subsystem** levels, as depicted in *Figure 2.1*:

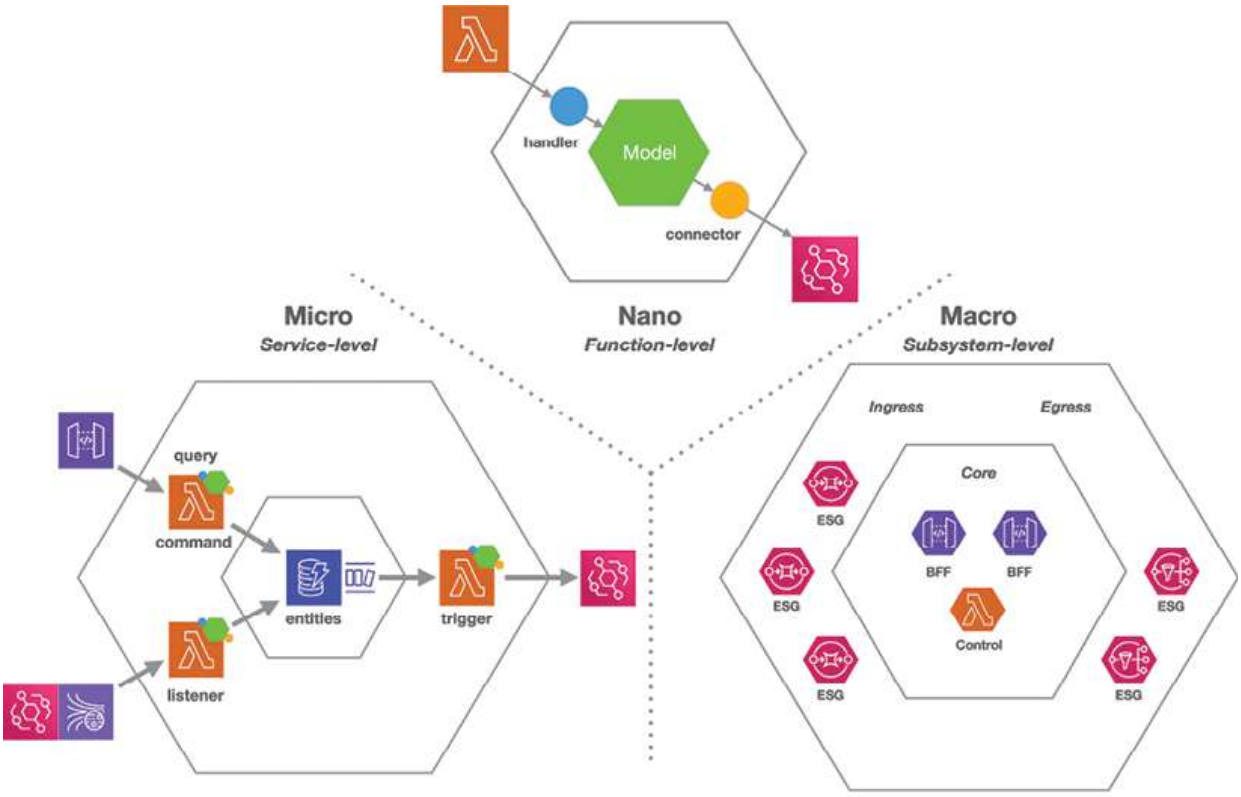


Figure 2.1: Hexagonal architecture

This *summary* diagram features detailed diagrams for each of the three levels, **Nano**, **Micro**, and **Macro**, all displayed together so we can get a bird's-eye view of how our serverless systems fit together. We need a mental picture of what is happening in a serverless *function*, how we combine these functions into autonomous *services*, and how we compose autonomous *subsystems* out of these services. We will see these diagrams again, individually, when we dig into the details. For now, let's review summary descriptions of each level.

Note that you can find a legend for all the icons in the preface of the book.

Function-level (nano)

The **function-level** or **nano hexagonal architecture** describes the structure and purpose of the *code* within a *serverless function*. This level is most like traditional hexagonal architecture, but we scale it down to the scope of an individual serverless function. The **handler** and **connector** *adapt* the **Model** to the cloud services. We will dig deeper into this level in the *Dissecting autonomous services* section of this chapter.

Service-level (micro)

The **service-level** or **micro hexagonal architecture** describes the structure and purpose of the *resources* within an *autonomous service*. This level is less traditional because we are spreading the code across multiple serverless functions. The **entities** of the internal domain model live in a dedicated datastore so that we can share them across all the functions. The **listener** and **trigger** functions *adapt* the internal domain model to the *domain events* exchanged between services. The **command** and **query** functions *adapt* the model for frontend communication. We will dig deeper into the details of this level in the *Dissecting autonomous services* section later in this chapter.

Subsystem-level (macro)

The **subsystem-level** or **macro hexagonal architecture** describes the structure and purpose of the autonomous services within an *autonomous subsystem*. This level is different from traditional hexagonal architecture because the adapters are full services instead of just code artifacts. The **Core** of the subsystem is composed of **Backend for Frontend** (**BFF**) services and **Control** services that work together to implement the *domain model* and the *internal domain events* exchanged between these services define the *ports* of the model. The **Ingress** and **Egress External Service Gateway** (**ESG**) services *adapt* external domain events to internal domain events. We will cover external domain events in the *Creating subsystem bulkheads* section of this chapter. We will dig into the details of this level when we cover the ESG pattern in *Chapter 7, Bridging Intersystem Gaps*.

Now that we have cover the foundational concepts, let’s learn how event-first thinking helps us create evolutionary systems.

Thinking about events first

In the first chapter, we covered a brief history of software integration styles and the forces that impact lead times. We designed autonomous services to enable teams to maximize their pace of innovation because they give teams the confidence they need to minimize lead time and batch size. However, to deliver on this promise, we need to change the way we act, which means we need a different way of thinking.

We need to do the following:

1. Start with event storming
2. Focus on verbs instead of nouns

3. Treat events as facts instead of ephemeral messages
4. Turn APIs inside out by treating events as contracts
5. Invert responsibility for invocation
6. Connect services through an event hub

In other words, we need to think **event-first**. We can start to change our perspective by using a technique called **event storming**.

Start with event storming

Event storming is a workshop-oriented technique that helps teams discover the behavior of their business domain. It begins with brainstorming. The team starts by coalescing an initial set of domain events on a board using orange sticky notes. Next, we sequence the cards to depict the flow of events.

The following is a simplified example of the flow of events for a typical food delivery service. I will use and elaborate on this example throughout the book:

Figure 2.2: Event storming – the flow of events

Along the way, the team will iteratively add more details using sticky notes of different colors, such as the following:

- The command (blue) that performed the action that generated the event

- The users (yellow) or external systems (pink) that invoked the command
- The aggregate business domain (tan) whose state changed
- The read-only data (green) that we need to support decision-making
- Any policies (gray) that control behavior
- The overall business process (purple) that is in play

Note that event storming is not a substitute for user stories and story mapping. User stories and story mapping are project management techniques for dividing work into manageable units and creating roadmaps. Event storming facilitates the discovery of user stories and the boundaries within our software architecture.

Focus on verbs instead of nouns

The flow of events discovered in the event-storming exercise clearly captures the behavior of the system. This event-first way of thinking is different because it zeroes in on the *verbs* instead of the *nouns* of the business domain. Conversely, more traditional approaches, such as object-oriented design, tend to focus on the nouns and create a class for each noun. However, *when we focus on the nouns, we tend to create services that are resistant to change* because they violate the SRP and ISP principles.

As an example, it is not uncommon to find a service whose *single responsibility* is everything to do with a single domain aggregate. These services will end up containing all the commands that operate on the data of the domain. However, as we discussed in the *SOLID principles* section, the SRP is intended to focus on the actors of the system. Different actors initiate different commands, which means that these noun-focused services

ultimately serve many masters with competing demands. This will impede our ability to change these services when necessary.

Instead, we need to *segregate the various commands across the different actors*. By focusing on the *verbs* of the domain model, we are naturally drawn to creating services for the different actors that perform the actions. This eliminates the competing demands that add unnecessary complexity to the code and avoids coupling an actor to unneeded commands.

Of course, now that the actors are the focal point of our services, we will need a way to share the nouns (that is, domain aggregates) between services without increasing coupling. We need a record of truth. To address this, we first need to start thinking of events as facts, instead of just ephemeral messages.

Treat events as facts instead of ephemeral messages

Let's recognize that when we think about events, we are focusing on the outputs of the system instead of the inputs. We are thinking in the past tense and thus we are focusing on the *facts* the system will produce over time. This is powerful in multiple ways.

It turns out we are implicitly building business analytics and observability characteristics into the system. For example, we can count the **ViewedMenu** events to track the popularity of the different restaurants and we can monitor the rate of **PlacedOrder** events to verify the health of the system.

We can also use this information to validate the hypothesis of each lean experiment we perform to help ensure we are building the right system and delivering on our business goals and objectives. In other words, event-first

thinking facilitates observability mechanisms that help build team confidence and thus momentum.

However, to turn events into facts, we must treat them as *first-class* citizens instead of ephemeral messages. This is different from traditional messaging-based architectures, where we throw away the messages once we have processed them. We don't want to treat events as ephemeral messages because we lose valuable information that we cannot easily recreate, if at all.

We will instead treat events as *immutable* facts and store them in an *event lake* in perpetuity. The event lake will act as the record of truth for the facts of the system. However, to make the record of truth complete we must think of events as contracts instead of mere notifications.

Turn APIs inside out by treating events as contracts

Many event-driven systems use events for notifications only. These *anemic events* only contain the identifier of the business domain entity that produced the event. Downstream services must retrieve the full data when they need it. This introduces coupling because it requires a synchronous call between the services. It may also create unwanted race conditions since the data can change before we retrieve it.

The usefulness of notification events as the record of truth is very limited because we will often refer to these facts far off in the future, well after the domain data has changed. To fully capture the facts, we need events to represent a **snapshot in time** of the state of the domain aggregate when the event occurred. This allows us to treat the facts as an audit log that is analogous to the transaction log of a database. This is a very powerful

concept because a database uses the transaction log as the record of truth to manage the state and integrity of the database.

We are essentially turning the database inside out and creating a **systemwide record of truth** that we can leverage to manage the state and integrity of the entire system. For example, we can leverage the facts to transfer (that is, replicate or rebuild) the state of domain aggregates (that is, nouns) between services. This eliminates the need for aligning services around domain aggregates and results in an immensely scalable and resilient system.

This means that we are turning our APIs inside out by **using events as the contracts between services**. This also implies a guarantee of backward compatibility, and we will therefore create strong contracts between services within a subsystem and even stronger contracts between subsystems. At first glance, it may appear that this way of thinking will make the system more rigid. In reality, we are making the system more flexible and evolutionary by inverting responsibility to downstream services so they can react to events as they see fit.

Invert responsibility for invocation

The DIP, as we covered earlier in the chapter, was a major advancement in software design, because it decoupled high-level policy decisions from low-level dependency decisions. This gave teams the flexibility to substitute different implementations of the low-level components without breaking the logic in the high-level components. In other words, the DIP facilitated the use of the **LSP** and the **OCP** to make systems much more stable and flexible.

We elevated the DIP to the architectural level by using events as the abstraction (that is, contract) between autonomous services. This promotes the stability of the system when we modify a service because we are holding the contracts constant to control the scope and impact of any given deployment. But we gain more than just stability; we also gain *flexibility*. The use of events for inter-service communication gives rise to an **inversion of responsibility** that makes systems **reactive**. The best way to understand this improvement is to compare the old *imperative* approach to the new *reactive* approach.

The traditional, *imperative* approach to implementing systems is *command* focused. One component determines when to invoke another. For example, in our food delivery system, we would traditionally have the checkout functionality make a synchronous call to an order management service to invoke a command that submits the customer's order. This means that we are coupling the checkout component to the presence of an order management service because it is responsible for the decision to invoke the command. This may not seem like a problem until we apply the same approach to retrieving driver status. We will need to invoke a service to retrieve driver status over and over again by any number of components and it will likely become a bottleneck.

Alternatively, we end up with a much more resilient and flexible system when we employ the *reactive* approach. The checkout component simply produces an **OrderPlaced** event. The order management service is now responsible for the decision to consume this event and react as it sees fit. The driver service simply produces **DriverStatusChanged** events when there is something useful to report. Any other service can take responsibility for reacting to driver events without impacting the driver service.

This *inversion of responsibility* is a key characteristic of autonomous services. It greatly reduces the complexity of the individual services because it reduces their responsibilities. A service is already aware of its own state, and it can simply produce events to reflect the changes without taking responsibility for what happens next. Downstream services take responsibility for how they react to upstream events. This completely decouples services from one another. They are all autonomous. This simplicity makes it much easier for teams to gauge the correctness and impact of any given change. Teams can be confident that the system will remain stable, if they uphold the contracts.

The **reactive** nature of event-first thinking is a paradigm shift, but the benefits are well worth the effort. A system becomes free to **evolve in unforeseen ways by simply adding consumers**. We can implement services in virtually any order because we can simulate upstream events and there is no coupling to downstream consumers. We gain the ability to create end-to-end test suites that don't require other services to be running at the same time. The bottom line is that the reactive nature of autonomous services enables autonomous teams to react much more quickly to feedback as they learn from their experiments.

Connect services through an event hub

There is a myth that event-driven systems are much more complex, but this couldn't be further from the truth. Event-first thinking allows us to create arbitrarily complex systems by connecting subsystems in a simple fractal topology as *Figure 2.3* depicts:

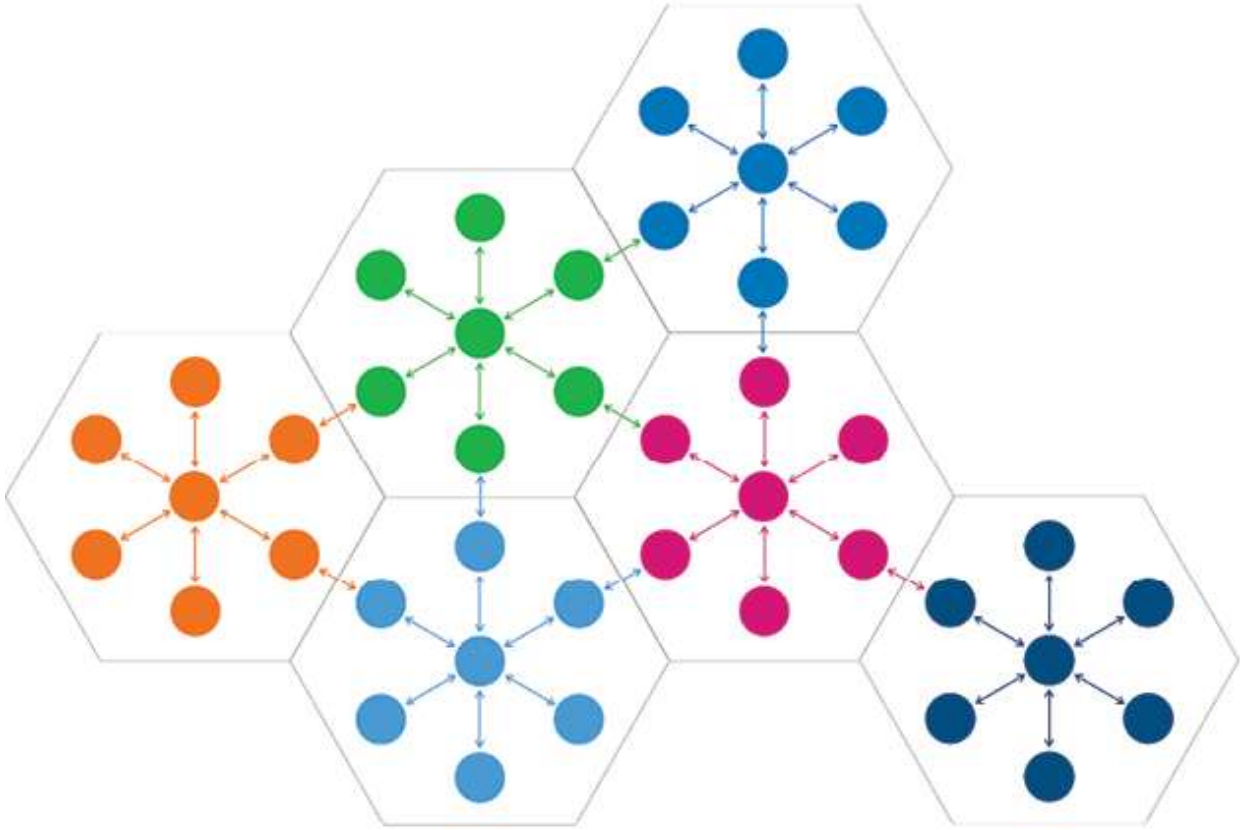


Figure 2.3: Event-first topology

At the heart of our event-first architecture is the **event hub**. It connects everything together and pumps events through the system. Each subsystem has an event hub at the center, and each autonomous service connects to the event hub through well-defined ports (that is, events).

From a single service to many services in a subsystem, and from a single subsystem to many subsystems in a system, this simple pattern of connecting autonomous services and subsystems by producing and consuming events repeats ad infinitum. This flexibility frees us to build ever-evolving systems. We will dig into the details of the event hub in *Chapter 4, Trusting Facts and Eventual Consistency*, and we will see how to connect subsystems in *Chapter 7, Bridging Intersystem Gaps*.

Event-first is a very powerful approach but adopting this way of thinking can be a journey. Let’s continue that journey by learning how to divide a system into autonomous subsystems, then we will move on to the autonomous service patterns within each subsystem, and finally, we will dig into the anatomy of these services. Then we will be ready to bring all the details together throughout the remaining chapters.

Dividing a system into autonomous subsystems

The goal of software architecture is to define boundaries that enable the components of the system to change independently. We could dive straight down into defining the individual services of the system, but as the number of services grows the system will become unwieldy. Doing so contributes to the creation of microliths and microservice death stars. As architects, our job is to facilitate change, which includes architecting a manageable system.

We need to step back and look at the bigger picture. We must break a complex problem down into ever-smaller problems that we can solve individually and then combine into the ultimate solution. We need to divide the system into a manageable set of high-level *subsystems* that each has a single reason to change. These subsystems will constitute the major *bounded contexts* of the system. We will apply the **SRP** along different dimensions to help us arrive at boundaries that enable change. This will facilitate *organizational scale* with separate groups managing the individual subsystems.

We also need our subsystems to be autonomous, in much the same way that we create autonomous services. This will give autonomous organizations the confidence to continuously innovate within their subsystems. We will

accomplish this by creating bulkheads between the subsystems. A system resembling the event-first topology depicted in *Figure 2.3* will begin to emerge. The purpose of each subsystem will be clear, and the subsystem architecture will allow the system to evolve in a dynamic business environment.

Let's look at some ways we can divide up a system.

By actor

A logical place to start carving up a system into subsystems is along the external boundaries with the external actors. These actors are the users and the external systems that directly interact with the system. Following the SRP, each subsystem might be responsible to one and only one actor.

In our event storming example earlier, we identified a set of domain events for a food delivery system. During the event storming workshop, we would also identify the users (yellow) and external systems (pink) that produce or consume those events, such as in *Figure 2.4*. In this example, we might have a separate subsystem for each category of the user: **Customer**, **Driver**, and **Restaurant**. We may also want a subsystem for each category of the external system, such as relaying orders to the restaurant's ordering systems, processing payments, and pushing notifications to customers:

Figure 2.4: System context diagram

Of course, this is a simple example. Enterprise systems may have many kinds of users and lots of external systems, including legacy and third-party systems. In this case, we will need to look for good ways to organize actors into cohesive groups and these groups may align with the business units.

By business unit

Another good place to look for architectural boundaries is between business units. A typical organizational chart can provide useful insights. Each unit will ultimately be the business owner of its subsystems and thus they will have a significant impact on when and how the system changes.

Keep in mind that we are interested in the organization of the company, not the IT department. **Conway's Law** states *organizations are constrained to produce designs which are copies of their communication structure*. We have seen that the communication structure leads to dependencies, which increases lead time and reduces the pace of innovation. So, we want to align each autonomous subsystem with a single business unit. We often refer to this approach as the *Inverse Conway Maneuver*.

However, the organizational structure of a company can be unstable. A company may reorganize its business units for a variety of reasons. So, we should look deeper into the work the business units actually perform.

By business capability

Ultimately, we want to draw our architectural boundaries around the actual business capabilities that the company provides. Each autonomous

subsystem should encapsulate a single business capability or at most a set of highly cohesive capabilities.

Going back to our event-storming approach and our event-first thinking, we are looking for logical groupings of related events (that is, verbs). There will be high temporal cohesion within these sets of domain events. They will be initiated by a group of related actors that are working together to complete an activity. For example, in our food delivery example, a driver may interact with a dispatch coordinator to ensure that a delivery is successful.

The key here is that the temporal cohesion of the activities within a capability helps to ensure that the components of a subsystem will tend to change together. This cohesion allows us to scale the SRP to the subsystem level when there are many different actors. The individual services within a subsystem will be responsible to the individual actors, whereas a subsystem is responsible to a single set of actors that work together in a business process (purple) to deliver a business capability:

Figure 2.5: Capabilities subsystems

Figure 2.5 depicts our food delivery system from the capabilities perspective. It is similar to the system context diagram in *Figure 2.4*, but the functionality is starting to take shape. However, we may find more subsystems when we look at the system from the perspective of the data life cycle.

By data life cycle

Another place to look for architectural boundaries is along the data life cycle. Over the course of the life of a piece of data, the actors that use and interact with the data will change and so will their requirements. Bringing the data life cycle into the equation will help uncover some overlooked subsystems. We will usually find these subsystems near the beginning and the end of the data life cycle. In essence, we are applying the SRP all the way down to the database level. We want to discover all the actors that interact with the data so that we can find all the actors and isolate these sources of change into their own bounded contexts (that is, an autonomous subsystem).

Going back to event-first thinking, we are stepping back and taking a moment to focus on the nouns (that is, domain aggregates) so that we can discover more verbs (that is, domain events) and the actors that produce those events. This will help find what I refer to as **slow data** (green). We typically zero in on the **fast data** (tan). This is the transactional data in the system that actors are continuously creating. However, the transactional data often relies on reference data and government regulation may impose records management requirements on how long we must retain the transactional data. We want to decouple these sources of change so that they do not impact the flexibility and performance of the transactional and analytics data. We will cover this topic in detail in *Chapter 5, Turning the Cloud into the Database*:

Figure 2.6: Data life cycle subsystems

Figure 2.6 depicts subsystems from the data life cycle perspective. We will likely need a subsystem upstream that owns the master data model that all downstream subsystems use as **reference data**. And all the way downstream we will usually have an **analytics** subsystem and a **records management** subsystem. In the middle lies the transactional subsystems that provide the capabilities of the system, like we saw in *Figure 2.5*. We will also want to carve out subsystems for any legacy systems.

By legacy system

Our legacy systems are a special case. In *Chapter 7*, *Bridging Intersystem Gaps*, and *Chapter 13*, *Don't Delay, Start Experimenting*, we will cover an event-first migration pattern for integrating legacy systems known as the **Strangler** pattern. Without going into detail here, we are creating an anti-corruption layer around the legacy systems that enable them to interact with the new system by producing and consuming domain events. This creates an evolutionary migration path that minimizes risk by keeping the legacy systems active and synchronized until we are ready to decommission them. This extends the substitution principle to the subsystem level because we can simply remove the legacy subsystem once the migration is complete.

We are essentially treating the legacy systems as an autonomous subsystem with bulkheads that we design to eliminate coupling between the old and the new and to protect the legacy infrastructure by controlling the attack surface and providing backpressure. We will use the same bulkhead techniques we are using for all subsystems: separate cloud accounts and external domain events.

Let's look at how we can create these subsystem bulkheads next.

Creating subsystem bulkheads

What is the most critical subsystem in your system? This is an interesting question. Certainly, they are all important, but some are more important. For many businesses, the customer-facing portion of the system is the most important. After all, we must be able to engage with the customers to provide a service for them. For example, in an e-commerce system, the customer must be able to access the catalog and place orders, whereas the authoring of new offers is less crucial. So, we want to protect the critical subsystems from the rest of the system.

Our aim is to fortify all the architectural boundaries in a system so that autonomous teams can forge ahead with experiments, confident in the knowledge that the blast radius will be contained when teams make mistakes. At the subsystem level, we are essentially enabling autonomous organizations to manage their autonomous subsystems independently. Let’s look at how we can fortify our autonomous subsystems with bulkheads.

Separate cloud accounts

Cloud accounts form natural bulkheads that we should leverage as much as possible to help protect us from ourselves. Far too often we overload our cloud accounts with too many unrelated workloads, which puts all the workloads at risk. At a bare minimum, development and production environments must be in separate accounts. But we can do better by having separate accounts, per subsystem, per environment. This will help control the blast radius when there is a failure in one account. Here are some of the natural benefits of using multiple accounts:

- We control the technical debt that naturally accumulates as the number of resources within an account grows. It becomes difficult, if not impossible, to see the forest for the trees, so to speak, when we put too many workloads in one account. The learning curve increases because the account is not clean. Tagging resources helps, but they are prone to omission. Engineers eventually resist making any changes because the risk of making a mistake is too high. The likelihood of a catastrophic system failure also increases.
- We improve our security posture by limiting the attack surface of each account. Restricting access is as simple as assigning team members to the right role in the right account. If a breach does occur, then access is limited to the resources in that one account. In the case of legacy systems, we can minimize the number of accounts that have access to the corporate network, preferably to just one per environment.
- We have less competition for limited resources. Many cloud resources have soft limits at the account level that throttle access when transaction volumes exceed a threshold. The likelihood of hitting these limits increases as the number of workloads increases. A denial-of-service attack or a runaway mistake on one workload could starve all other workloads. We can request increases to these limits, but this takes time. Instead, the default limits may provide plenty of headroom once we allocate accounts for individual subsystems.
- Cost allocation is simple and error resistant because we allocate everything in an account to a single cost bucket without the need for tagging. This means that no unallocated costs occur when tagging is incomplete. We also minimize the hidden costs of orphaned resources because they are easier to identify.

- Observability and governance are more accurate and informative because monitoring tools tag all metrics by account. This allows filtering and alerting by subsystem. When failures do occur, the limited blast radius also facilitates root cause analysis and a shorter mean time to recovery.

Having multiple accounts means that there are cross-cutting capabilities that we must duplicate across accounts, but we will address this shortly when we discuss automation in the *Governing without impeding* section.

External domain events

We have already discussed the benefits of using events as contracts, the importance of backward compatibility, and how asynchronous communication via events creates a bulkhead along our architectural boundaries. Now we need to look at the distinction between internal and external domain events.

Within a subsystem, its services will communicate via **internal domain events**. The definitions of these events are relatively easy to change because the autonomous teams that own the services work together in the same autonomous organization. The event definitions will start out messy but will quickly evolve and stabilize as the subsystem matures. We will leverage the Robustness principle to facilitate this change. The events will also contain raw information that we want to retain for auditing purposes but that is of no importance outside of the subsystem. All of this is OK because it is *all in the family*, so to speak.

Conversely, across subsystem boundaries, we need more regulated team communication and coordination to facilitate changes to these contracts. As we have seen, this communication increases lead time, which is the opposite

of what we want. We want to limit the impact that this has on internal lead time, so we are free to innovate within our autonomous subsystems. We essentially want to hide internal information and not *air our dirty laundry in public*.

Instead, we will perform all inter-subsystem communication via **external domain events** (sometimes called *integration events*). These external events will have much more stable contracts with stronger backward compatibility requirements. We will intend for these contracts to change slowly to help create a bulkhead between subsystems. **Domain-Driven Design** (**DDD**) refers to this technique as **context mapping**, such as when we use domain aggregates with the same terms in multiple bounded contexts, but with different meanings.

External events represent the subsystem's *ports* in hexagonal terminology. In *Chapter 7*, *Bridging Intersystem Gaps*, we will cover the **External Service Gateway** (**ESG**) pattern. Each subsystem will treat related subsystems as external systems. We will bridge the internal event hubs of related subsystems to create the event-first topology depicted in *Figure 2.3*. Each subsystem will define *egress* gateways that define what events it is willing to share and hide everything else. Subsystems will define *ingress* gateways that act as an anti-corruption layer to consume upstream external domain events and transform (that is, adapt) them to its internal formats.

Now that we have an all-important subsystem architecture with proper bulkheads, let's look at the architecture within a subsystem. Let's see how we can decompose an autonomous subsystem into autonomous services.

Dissecting an autonomous subsystem

At this point, we have divided our system into autonomous subsystems. Each subsystem is responsible to a single dominant actor who drives change. All subsystems are autonomous because they communicate via external domain events, and they are each housed in a separate cloud account that forms a natural bulkhead. This autonomy allows us to change the subsystems independently.

Now we are ready to start decomposing our subsystems into autonomous services. Again, the SRP plays a major role in defining the boundaries within a subsystem. First, we need to place a subsystem in context, then we will set up common components, and finally, we apply the major autonomous service patterns.

Context diagram

We will apply a set of autonomous service patterns to decompose a subsystem into services. These patterns cater to the needs of different categories of actors. So, we need to understand the context of an autonomous subsystem before we can decompose it into autonomous services. In other words, we need to know all the external actors that the subsystem will interact with. During event storming, we identified the behavior of the system and the users and external systems that are involved. Then we divided the system into autonomous subsystems so that each has a *single axis of change*.

A simple context diagram can go a long way to putting everyone on the same page regarding the scope of a subsystem. The **context diagram** enumerates

all the subsystem's external actors using *yellow* cards for *users* and *pink* cards for *external systems*. The diagram will contain a subset of the actors identified during event storming. We have encapsulated many of the original actors within other subsystems, so we will treat those subsystems as external systems. *Figure 2.7* depicts the context of the **Customer** subsystem:

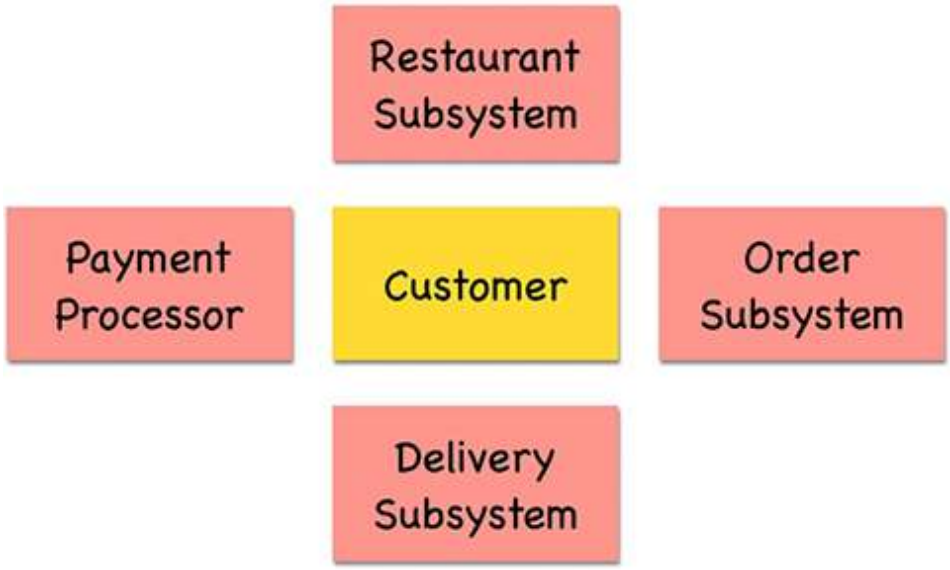


Figure 2.7: Subsystem context diagram

The **Customer** subsystem of our example Food Delivery System might have the following actors:

- The **Customer** will be the user of this subsystem and the dominant actor.
- The **Restaurant Subsystem** will publish external domain events regarding the restaurants and their menus.
- A **Payment Processor** must authorize the customer's payment method.
- The subsystem will exchange **OrderPlaced** and **OrderReceived** external domain events with the **Order Subsystem**.

- The **Delivery Subsystem** will publish external domain events about the status of the order.

Now that the context is clear, we can start decomposing the system into its frontend, services, and common components.

Micro frontend

Each autonomous subsystem is responsible to a single primary user or a single cohesive group of users. These users will need a main entry point to access the functionality of the subsystem. Each subsystem will provide its own independent entry point so that it is not subject to the changing requirements of another subsystem.

The user interface will not be monolithic. We will implement the frontend using autonomous micro-apps that are independently deployed. The main entry point will act as a metadata-driven assembly and menu system. This will allow each micro-app to have a different reason to change and help ensure that the frontend is not responsible for increasing lead times and impeding innovation.

We will cover the frontend architecture in detail in *Chapter 3*, *Taming the Presentation Tier*.

Event hub

Each autonomous subsystem will contain its own independent event hub, as depicted in *Figure 2.8*, to support asynchronous inter-service communication between the autonomous services of the subsystem. Services will publish domain events to the event hub as their state changes. The event hub will receive incoming events on a bus. It will route all events to the event lake for

storage in perpetuity, and it will route events to one or more channels for consumption by downstream services:

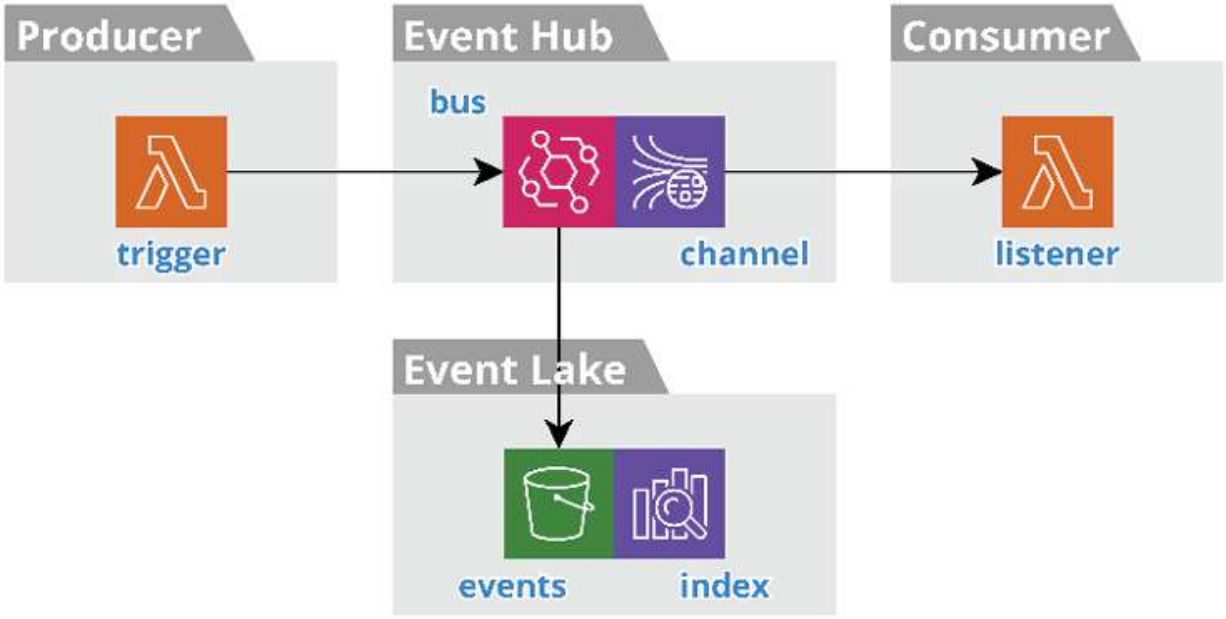


Figure 2.8: Event hub

We will cover the event hub in detail in *Chapter 4, Trusting Facts and Eventual Consistency*. In *Chapter 7, Bridging Intersystem Gaps*, we will cover how to bridge the event hubs of different subsystems together to create the event-first topology depicted in *Figure 2.3*.

Autonomous service patterns

There are three high-level autonomous service patterns that all our services will fall under as depicted in *Figure 2.9*. At the boundaries of our autonomous subsystems are the **Backend For Frontend** (**BFF**) and **External Service Gateway** (**ESG**) patterns. Between the boundary patterns lies the **Control** service pattern. Each of these patterns is responsible to a different kind of actor, and hence supports different types of changes:

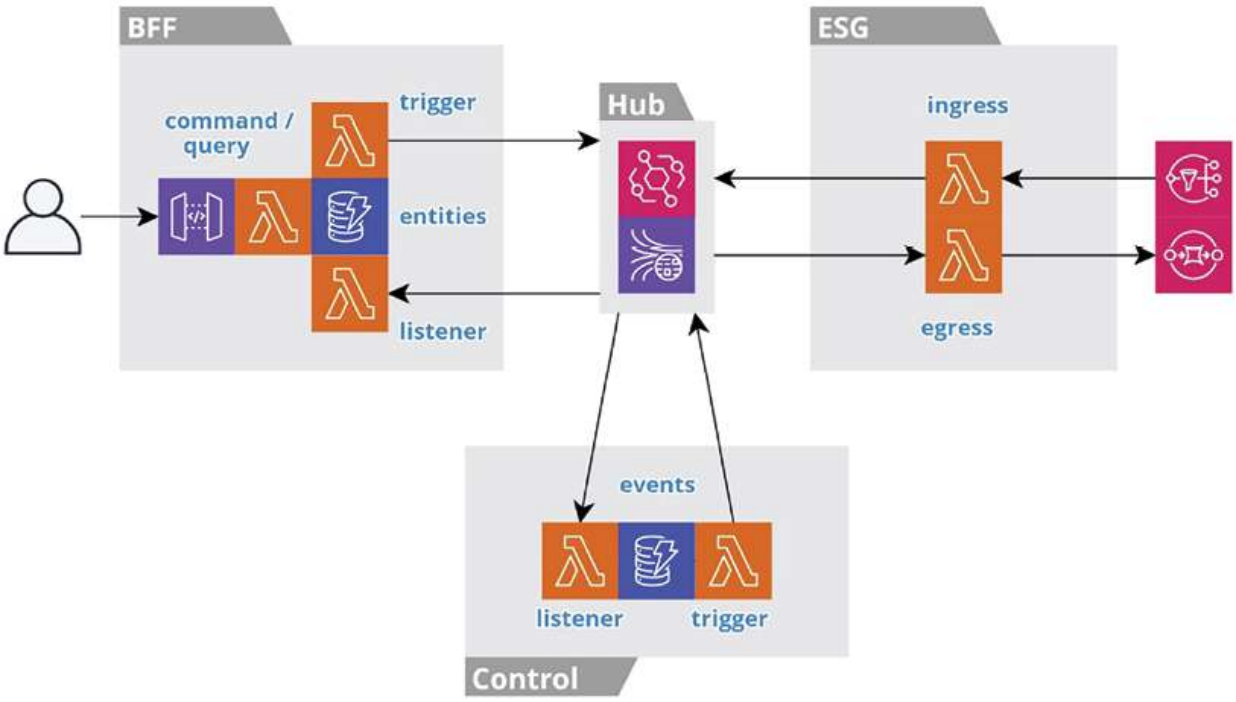


Figure 2.9: Service patterns

Backend For Frontend

The **Backend For Frontend** (**BFF**) pattern works at the boundary of the system to support end users. Each BFF service supports a specific frontend micro-app, which supports a specific actor.

In the **Customer** subsystem of our example Food Delivery System, we might have BFFs to browse restaurants and view their menus, sign up and maintain account preferences, place orders, view the delivery status, and view order history. These BFFs typically account for about 40% of the services in a subsystem.

A **listener** function consumes domain events from the event hub and caches **entities** in materialized views that support queries. The synchronous API provides **command** and **query** operations that support the specific user

interface. A **trigger** function reacts to the mutations caused by commands and produces domain events to the event hub.

We will cover this pattern in detail in *Chapter 6*, *A Best Friend for the Frontend*.

External Service Gateways

The **External Service Gateway** (**ESG**) pattern works at the boundary of the system to provide an anti-corruption layer that encapsulates the details of interacting with other systems, such as third-party, legacy, and sister subsystems. They act as a bridge to exchange events between the systems.

In the **Customer** subsystem of our example Food Delivery System, we might have ESGs to receive menus from the **Restaurant** subsystem, forward orders to the **Order** subsystem, and receive the delivery status from the **Delivery** subsystem. The **Order** subsystem would have ESGs that integrate with the various order systems used by restaurants. The **Delivery** subsystem would have an ESG to integrate with a push notifications provider. These ESGs typically account for upwards of 50% of the services in a subsystem.

An **egress** function consumes internal events from the event hub and then transforms and forwards the events out to the other system. An **ingress** function reacts to external events in another system and then transforms and forwards those events to the event hub.

We will cover this pattern in detail in *Chapter 7*, *Bridging Intersystem Gaps*.

Control services

The **Control Service** pattern helps minimize coupling between services by mediating the collaboration between boundary services. These services

encapsulate the policies and rules that are governed by the business owners. They are completely asynchronous. They consume events, perform logic, and produce new events to record the results and trigger downstream processing.

We use these services to perform complex event processing and orchestrate business processes. They leverage the systemwide event sourcing pattern and rely on the ACID 2.0 properties (*Associative, Commutative, Idempotent, and Distributed*). In the **Delivery** subsystem of our example Food Delivery System, we might have a control service that implements a state machine to orchestrate the delivery process under many different circumstances. Control services typically account for about 10% of the services in a subsystem.

A **listener** function consumes lower-order events from the event hub and correlates and collates them in a micro **events** store. A **trigger** function applies rules to the correlated events and publishes higher-order events back to the event hub.

We will cover this pattern in detail in *Chapter 8, Reacting to Events with More Events*.

Now, let’s look at the anatomy of an autonomous service.

Dissecting an autonomous service

Up to this point, we have discussed using the SRP as a guide for defining architectural boundaries that help ensure the system has the flexibility to change with the needs of the different actors. We have covered dividing a system into autonomous subsystems and decomposing an autonomous

subsystem into autonomous services. Now we move on to the anatomy of an individual autonomous service.

Each autonomous team has the ultimate responsibility for making the decisions that are best for the services they own. Embracing a polyglot-everything mindset and empowering the teams to make these decisions gives them the freedom they need to maximize innovation. Still, every service needs a starting point to jump-start the process of discovery and continuous improvement. The following sections cover the common elements that go into the implementation of autonomous services.

You can find a service template here: https://github.com/jgilbert01/templates/tree/master/template-bff-service.

One of the most interesting things to note is that there is much more to a service than just its runtime code.

Repository

Each service has its own source code repository. This is due, in part, to the fact that modern distributed source control tools, such as Git, make it very easy to create and distribute new repositories. In addition, hosted offerings drive this point home by making the repository the focal point of their user experience. Furthermore, modern CI/CD pipelines tools assume that the repository is the unit of deployment. All these factors steer us towards this best practice.

Yet, the most important reason that each service has its own repository is autonomy. We want to drive down our lead times and sharing a repository with other teams will certainly cause friction and slow teams down. Separate

repositories also act as bulkheads and shield teams from mistakes made by other teams.

They also protect us from ourselves in that we cannot accidentally create a dependency on the source code owned by another team, just because it is in the same repository. Instead, we must purposefully and explicitly create shared libraries, as discussed below, that will have their own repositories and release cycles.

CI/CD pipeline and GitOps

Each service has its own CI/CD pipeline, as defined by a configuration file in the root of its repository. Modern CI/CD pipelines enhance the concept of GitOps, which is the practice of using Git pull requests to orchestrate the deployment of infrastructure.

The pipeline hooks into the state changes of the repository and pull requests to trigger and coordinate CI/CD activities. Each push to a repository triggers CI tests to ensure that the code is behaving as expected. The creation of a pull request triggers deployment to the non-production environment and signals that the code is ready for review. Approval of a pull request triggers a production deployment.

This is a very powerful approach that becomes even stronger when combined with the concepts and practices of decoupling deployment from release, multiple levels of planning, task branch flow, and regional canary deployments. We will cover this in detail in *Chapter 11, Choreographing Deployment and Delivery*.

Tests

Automated testing plays a vital role in giving teams the confidence to continuously deploy. To this end, test cases make up the majority of the code base for a feature and unit tests make up the majority of the test cases. We execute unit tests, integration tests, contract tests, and transitive end-to-end tests in the CI/CD pipeline in isolation from all external resources. We will cover testing in *Chapter 11*, *Choreographing Deployment and Delivery*.

Stack

We deploy each service as a set of cloud resources that we will refer to as a **stack**. We *declaratively* define the resources of a stack in a **serverless.yml** configuration file in the root of the repository. We use the **Serverless Framework** to initiate deployments and execute the cloud provider's deployment management service, such as AWS CloudFormation. The deployment management service manages the life cycle of the resources as a group. It compares the current state of the stack to the latest declarations and applies any changes. It adds new resources, updates existing resources, and deletes removed resources. And it deletes all resources when we delete the stack to ensure there are no orphaned resources. We use diagrams such as *Figure 2.10* to depict the main resources in a service (that is, a stack):

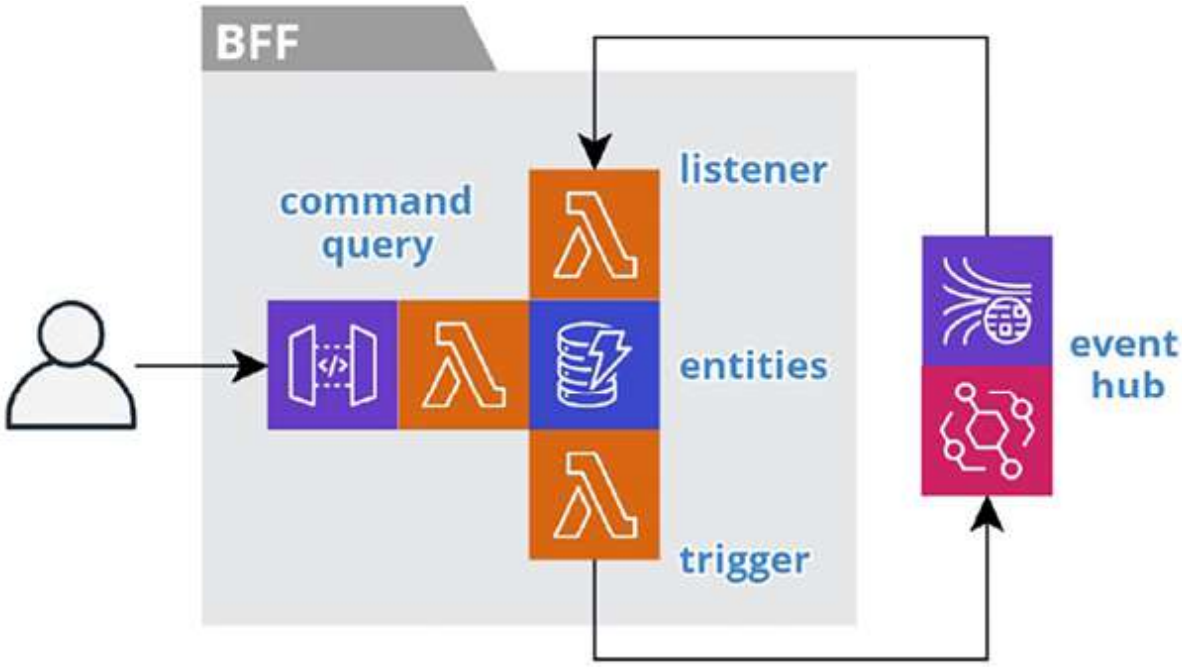


Figure 2.10: A typical service stack

The gray box logically equates to the **service** and physically equates to both the **repository** and the **stack**. The icons represent the cloud resources within the stack. We place icons next to each other to indicate communication. This results in tighter diagrams. The nearest arrow implies the flow of communication. A legend for all the icons is available in the preface.

We will cover more details of creating a stack in *Chapter 6, A Best Friend for the Frontend*, and *Chapter 11, Choreographing Deployment and Delivery*.

Persistence

Each service will own and manage its own data. Following *polyglot persistence* practices, each service will use the type of database that best supports its needs. These serverless resources are managed as part of the stack.

Services will consume domain events from upstream services and cache the necessary data as *lean* **materialized views**. The high availability of these serverless data stores creates an inbound bulkhead that ensures necessary data is available even when upstream services are not. This also greatly improves data access latency.

We will leverage the **Change Data Capture** (**CDC**) mechanism of a data store to **trigger** the publishing of domain events when the state of the data changes. We can also use CDC to control the flow of data within a service.

We will cover the details of the persistence layer in *Chapter 5, Turning the Cloud into the Database*.

Trilateral API

Each service will have up to **three APIs**: one for the events it consumes, another for the events it produces, and one for its synchronous interface. Not all these interfaces are required. Most services will consume events, but not all will publish events. For example, a BFF service that provides a read-only view of data would only consume events, cache the data, and provide a synchronous API to access the data. Control services and most ESGs do not have a synchronous interface.

Events

Following our event-first approach, the APIs for events are the most important, because they dictate how a service will interact with other services. A service should document the events it consumes and those that it produces. This could be as simple as a listing in the **README** file at the root of the repository. We can document the JSON structure of *internal domain events* using TypeScript interface notation. For *external domain*

events, a standard such as **OpenAPI** (https://www.openapis.org) or **JSON Schema** (https://json-schema.org) may be helpful. The cloud infrastructure may provide a registry service to capture the schemas of all the event types in the subsystem. You can also use a tool like **Event Catalog** (https://www.eventcatalog.dev) to make it easier to explore the producers and consumers in your subsystem. We will cover events in detail in *Chapter 4*, *Trusting Facts and Eventual Consistency*.

API Gateway

Services operating at the boundaries of the system, such as BFFs, will have a synchronous interface. We will implement these using an API Gateway.

We design the API of a BFF specifically for a single frontend micro-app. One team owns the frontend and the BFF, so official documentation for the API may not be necessary. However, we may use a self-documenting API, such as **GraphQL**. We will cover BFFs in detail in *Chapter 6, A Best Friend for the Frontend*.

Some ESG services will also require an API Gateway, such as implementing a webhook to receive events from a third-party system or providing an Open API for your own SaaS system. We will cover the ESG pattern in detail in *Chapter 7*, *Bridging Intersystem Gaps*.

Functions

We will implement the business logic of an autonomous service as serverless functions using the cloud provider's **Function-as-a-Service** (**FaaS**) offering, such as AWS Lambda. It is important to point out that while each function is *independent*, we will manage functions as a *group* within the **stack** that

owns all the *resources* of the **service**. To account for this distinction, we will use the **hexagonal architecture** that we introduced in the *Building on proven concepts* section.

We will architect our functions at two levels, the *nano* or *function level* and the *micro* or *service level*. In other words, we need to architect the code within each function and architect the functions within each service. Now let’s dig further into the nano and micro architecture of our serverless functions.

Nano architecture

At the **function** or **nano** level, we scale our **hexagonal architecture** down so that each serverless function cleanly executes a *fragment* of the business logic within an autonomous service. *Figure 2.11* depicts the structure and purpose of the code within an individual serverless function.

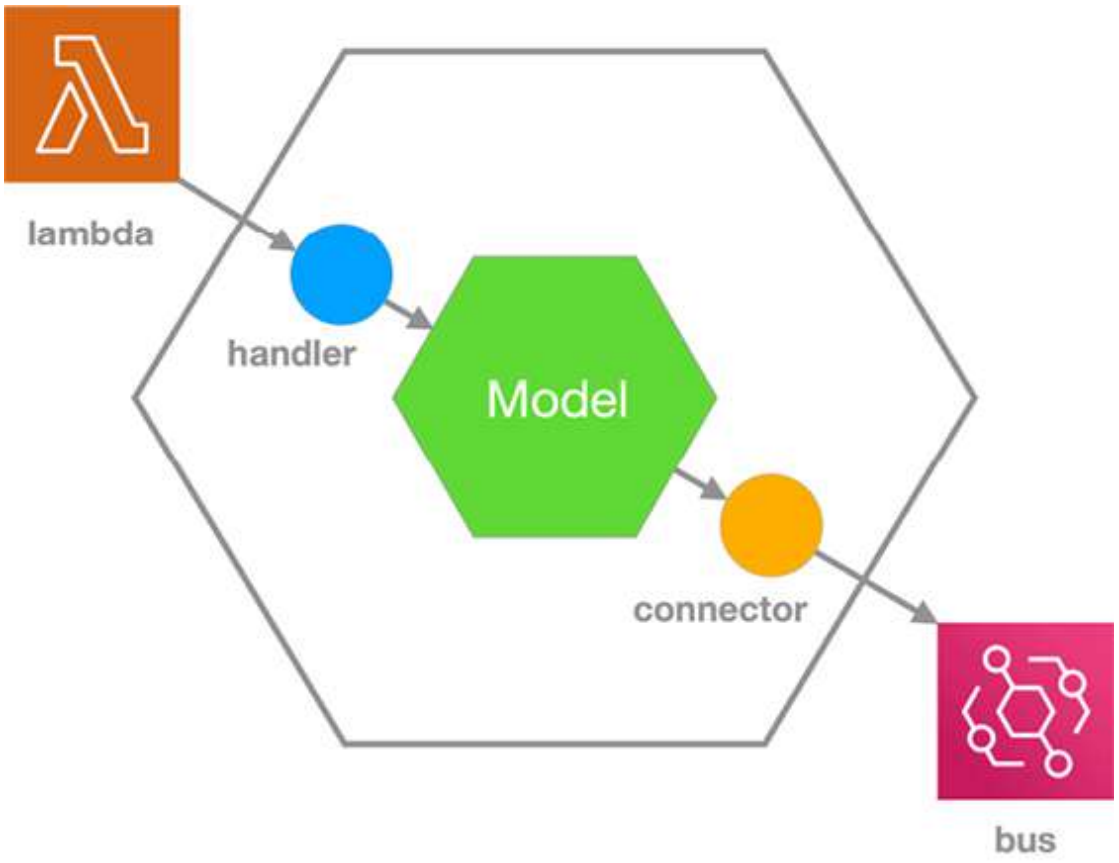


*Figure 2.11: Function-level – nano hexagonal architecture The FaaS service (that is, **lambda**) invokes a serverless function and dictates the signature of the input parameters. We do not want this signature to pollute our business logic. In turn, our business logic makes the outbound calls to cloud services, such as a **bus** or datastore. Again, we do not want the signatures of these cloud services to pollute our business logic. So, we will separate the business logic into a model and isolate these dependencies in adapters.*

We will implement the business logic (that is, the **Model**) as classes and functions that expose cloud-agnostic interfaces (that is, *ports*). We will implement a **handler** function that *adapts* the **lambda** signature to the model. The handler will inject the model with a **connector** class that *adapts* the model's outbound calls to the signature of the cloud service, such as the **bus**.

This *nano* architecture will allow us to easily move the *model* to a different runtime environment or even another cloud provider by *substituting* different implementations for the *handlers* and *connectors*. In *Chapter 11, Choreographing Deployment and Delivery*, we will see how this architecture facilitates a *serverless testing honeycomb*.

Micro architecture

At the **service** or **macro** level, we scale our **hexagonal architecture** to show how multiple serverless functions work together to implement the business logic of the autonomous service. *Figure 2.12* depicts the structure and purpose of the resources within an *autonomous service*.

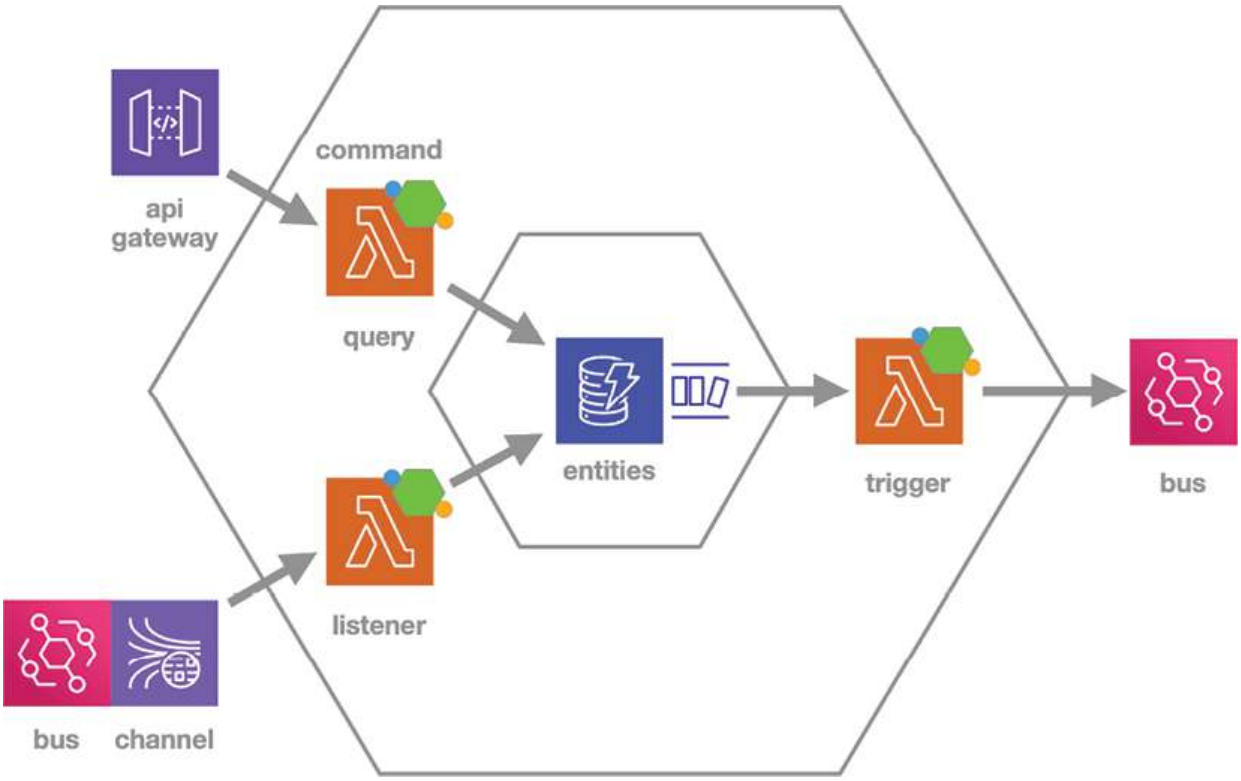


Figure 2.12: Service-level – micro hexagonal architecture This diagram presents an expanded format of the same BFF services depicted in condensed format in Figure 2.10 to highlight the different roles played by the various resources.

We store the internal domain model in a dedicated datastore (**entities**) so that we can share the data across all the functions. This data represents the internal domain *model* of the service, and that data format defines the interface (that is, *ports*). The serverless functions act as *adapters* to map between the internal model and the external model.

An autonomous service will typically have two to five serverless functions. Each function has its own *nano* architecture with the appropriate *handler* and *connector* based on the cloud services it interacts with, such as AWS Kinesis, AWS API Gateway, AWS DynamoDB, AWS DynamoDB Streams, or AWS EventBridge.

The **listener** function sits on the *driving* side because it consumes domain events from a **channel** and collects the data that is a prerequisite for the capabilities of the service. It *adapts* the data from the incoming domain events to the internal domain model and saves the results in the local **entities** datastore.

The **command**/**query** function sits on the *driving* side because it supports the user actions that drive work through the subsystem. The *nano model* implements the business logic for the queries and commands and the function *adapts* the model for frontend communication.

The **trigger** sits on the *driven* side because it reacts to the state changes within the service. It consumes internal change events from the datastore's CDC stream. It *adapts* the data from the change events to the domain event model and publishes the results.

The nano and micro levels work together to keep our architecture clean and decoupled. They enable the **Command, Publish, Consume, Query** (**CPCQ**) pattern that we use to connect services together to carry out the work of an autonomous subsystem. Work starts upstream. A user performs

an action, and a **command** function updates the state of the domain model. Then a **trigger** function publishes a domain event to record the fact that the action occurred and drive further processing. Downstream, one or more **listener** functions take responsibility for consuming the domain event and cache the needed data so that their users can **query** for the information they need to perform the next action. We repeat this simple pattern as many times as necessary to deliver the desired functionality of an *autonomous subsystem*.

We will see examples of *stream processor* functions in *Chapter 4, Trusting Facts and Eventual Consistency*. We will also see examples of *REST* and *GraphQL* functions in *Chapter 6, A Best Friend for the Frontend*.

Shared libraries

We will tend to shy away from using shared libraries for reusing business logic between services. This will help us avoid the false reuse that creates coupling and complexity. We will use open-source libraries for crosscutting concerns. Over time, duplicated logic can be refactored into libraries when it proves to not be false reuse. We will cover false reuse in *Chapter 13, Don't Delay, Start Experimenting.*

Now that we have defined our architectural boundaries, let's see how we can let go and govern without impeding innovation.

Governing without impeding

As architects, once we have defined the architectural boundaries of the system, we need to let go and get out of the way, unless we want to become an impediment to innovation. But letting go is difficult. It goes against our

nature; we like to be hands-on. And it flies in the face of traditional governance techniques. But we must let go for the sake of the business, whether the business realizes this or not.

Governance has an understandable reputation for getting in the way of progress and innovation. Although it has good intentions, the traditional manual approach to governance actually increases risk, instead of reducing it, because it increases lead time, which diminishes an organization's ability to react to challenges in a modern dynamic environment. But it doesn't have to be this way.

We have already taken major strides to mitigate the risks of continuous innovation. We define architectural boundaries that limit the scope of any given change, and we fortify these boundaries to control the blast radius when honest human errors happen. We do this because we know that *to err is human*. We know that mistakes are inevitable, no matter how rigorous a governance process we follow.

Instead of impeding innovations, we must empower teams with a culture and a platform that embraces continuous governance. This is a safety net that gives teams and management confidence to move forward, knowing that we can catch mistakes and make corrections in real time. Automation and observability are the key elements of continuous governance. Let's see how we can put this safety net in place and foster a culture of robustness.

Providing automation and cross-cutting concerns

A major objective of governance is to ensure that a system is compliant with regulations and best practices. These include the typical *-ilities*, such as

scalability and reliability, and of course security, along with regulations such as NIST, PCI, GDPR, and HIPAA. The traditional approach includes manual audits of the architecture. These gates are the reason governance has a reputation for impeding progress. They are labor intensive and worse yet; they are error prone.

Fortunately, we now have a better option. Our deployments are fully automated by our CI/CD pipelines. This is already a significant improvement in quality because **Infrastructure as Code** reduces human error and enables us to quickly fail forward. We still have some manual gates for each deployment.

The first gate is code review and approval of a pull request. We perform this gate quickly because each task branch has a small batch size. The second gate is the certification of a regional canary deployment. We deploy to one region for continuous smoke testing before deploying to other regions. We will cover CI/CD pipelines in detail in *Chapter 11, Choreographing Deployment and Delivery*.

We also have observability, which provides timely, actionable information so that we know when to jump into action and we can recover quickly. We will cover this in *Chapter 12, Optimizing Observability*. We will take automation further and harden our build processes by adding continuous auditing and securing the perimeter of our subsystems and our cloud accounts. We will cover these topics in *Chapter 10, Securing Autonomous Subsystems in Depth*.

However, these are all cross-cutting concerns, and we don’t want teams to reinvent these capabilities for each autonomous subsystem. We need a dedicated team with the knowledge and specialized skills to manage an integrated suite of SaaS tools, stamp out accounts with a standard set of

capabilities, and maintain these cross-cutting concerns for use across the accounts. Yet, the owners of each autonomous subsystem must have control over when to apply changes to their accounts and have the flexibility to override and/or enhance features as their circumstances dictate.

Even with these cross-cutting concerns in place, the reality is that many aspects of the approach and architecture are new and unfamiliar, so the next part of the governance equation is promoting a culture of robustness.

Promoting a culture of robustness

Our goal of increasing the pace of innovation leads us to a rapid feedback loop with small batch sizes and short lead times. We are deploying code much more frequently and these deployments must result in *zero downtime*. To eliminate downtime, we must uphold the contracts we have defined within the system. However, traditional *versioning* techniques fall apart in a dynamic environment with a high rate of change. Instead, we will apply the **Robustness principle**.

The Robustness principle states **be conservative in what you send, be liberal in what you receive**. This principle is well suited for continuous deployment, where we can perform a successive set of deployments to make a conforming change on one side of a contract, followed by an upgrade on the other side and then another on the first side to remove the old code. The trick is to develop a culture of robustness where this three-step dance is committed to team muscle memory and becomes second nature.

In *Chapter 11, Choreographing Deployment and Delivery*, we will cover a lightweight continuous delivery process that is geared for robustness. It

includes three levels of planning, GitOps, CI/CD pipelines, regional canary deployment, and more. It forms a simple automated bureaucracy that governs each deployment but leaves the order of deployments completely flexible.

In my experience, autonomous teams are eager to adopt a culture of robustness, especially once they get a feel for how much more productive and effective they can become. But this is a paradigm shift, and it is unfamiliar from a traditional governance perspective. Everyone must have the confidence to move at this pace. As architects, we need to be evangelists and promote this cultural change, both upstream and downstream. We need to educate everyone on how everything we are doing comes together to provide a safety net for continuous discovery.

Finally, let's see how metrics can guide governance.

Harnessing the four key team metrics

Observability metrics are an indispensable tool in modern software development. We cover this topic in detail in *Chapter 12*, *Optimizing Observability*. Autonomous teams are responsible for leveraging the observability metrics of their apps and services as a tool for self-governance and self-improvement. In my experience, teams truly value these insights and thrive on the continuous feedback.

From a systemwide governance perspective, we should focus our energy on helping teams that are struggling. In their book *Measure Software Delivery Performance with Four Key Metrics* (https://itrevolution.com/articles/measure-

software-delivery-performance-four-key-metrics), Nicole Forsgren, Gene Kim, and Jez Humble put forth four metrics that we can harness to help us identify which teams may need more assistance and mentoring:

- **Lead time**: How long does it take a team to complete a task and push the change to production?
- **Deployment rate**: How many times a day is a team deploying changes to production?
- **Failure rate**: How often does a deployment result in a failure that impacts a generally available feature?
- **Mean Time to Recovery** (**MTTR**): When a failure does occur, how long does it take the team to fail forward with a fix?

The answers to these questions clearly indicate the maturity of a specific team. We certainly prefer lead time, failure rate, and MTTR to be low and deployment rate to be high. Teams that are having trouble with these metrics are usually going through their own digital transformation and are eager to receive mentoring and coaching. We can collect metrics from our issue-tracking software and CI/CD tool and track them alongside all the others in our observability tool.

Summary

In this chapter, we learned how to define architectural boundaries that enable change. The key is understanding that people (that is, actors) are the source of change and then aligning our boundaries along these axes of change. We found that event-first thinking, with its focus on verbs and facts, naturally helps us with this alignment.

We brought this all together to divide a system into autonomous subsystems, and these into autonomous services. We saw how we can create arbitrarily complex systems following this simple fractal pattern of autonomous components. Then, we dissected the anatomy of autonomous services and learned how to leverage observability and automation to govern without impeding innovation.

In the next chapter, we will start digging into the details of our architecture. We will learn how the micro frontend approach, along with other new techniques, helps to bring the seemingly endless churn at the presentation layer under control.

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.

**Limited Offer*

3

Taming the Presentation Tier

In *Chapter 2, Defining Boundaries and Letting Go*, we defined the architecture of a system by dividing it into a set of autonomous subsystems. These architectural boundaries create bulkheads that enable change by promoting autonomy. They give autonomous teams the confidence to push the limits and continuously deliver business value because they know these fortified boundaries will help control the blast radius when things go wrong.

Now we turn our attention to the frontend user experience. Frontend architecture has been going through a revolution. Many of the new techniques are only at the beginning of the adoption curve. In this chapter, you will learn how to decompose the frontend into micro applications to enable change and ensure autonomy. You will also learn how to design for offline-first.

In this chapter, we're going to cover the following main topics:

- Presentation tier innovation – zigzagging through time
- Breaking up the frontend monolith
- Dissecting micro frontends
- Designing for offline-first

Presentation tier innovation – zigzagging through time

The presentation tier is a battleground. It always has been. It always will be. It is the most visible part of a system, so it elicits the most feedback and experiences the most churn. From a functional perspective, this is good, because it leads to innovations that help us zero in on the right solutions for users. From a technical perspective, this volatility is more of a mixed bag.

Today, a technical war rages over the use of UI frameworks, such as React, Vue, and Angular. It is good to have choices and the competition drives technical innovation that benefits everyone. However, it can also create a skills gap that impedes progress when team members move between projects that use different frameworks. We will provide some relief for this problem in the *Dissecting micro frontends* section, but I suspect we will never end the framework wars. However, debates over things such as client-side versus server-side rendering are more clear-cut. Let’s see how.

Client-side versus server-side rendering

I started my career as a Visual Basic programmer in the client-server programming area. But I wrote my share of mainframe green screens as well. So, I had a feel for the differences in both the developer and the user experience of the different rendering approaches.

Everyone preferred the client-side rendering experience, but it had its problems, such as versioning and installation headaches and the limits of hardware and network performance at the time. So, when the World Wide Web emerged, rendering moved back to the server side. However, the user experience suffered, and this drove the need for the **Asynchronous JavaScript and XML** (**Ajax**) innovation that moved some of the rendering

back to the client side. We referred to this back and forth between client-side and server-side rendering as a **pendulum**.

So, when **Single-Page Applications** (**SPAs**) and **Angular 1** arrived, my initial reaction was, “Here we go again, the pendulum is swinging back to the client side, and we are going to experience a similar set of problems again.” However, I had spent the last year wrangling with running server-side presentation tiers in the cloud, so I knew how costly and cumbersome it was to deliver a scalable solution.

This is when I realized that the pendulum wasn’t just swinging back and forth; it was zigzagging through time. With each cycle, the context changed. Different options or variations on options would emerge that changed how we weighed the different decision variables. And this time around, we had the option to deploy the presentation tier to the edge of the cloud.

I knew everything was going to change the moment I realized that I could easily deploy the SPA artifacts to an AWS S3 bucket and serve them through AWS CloudFront, with no need to manage any servers. With little to no elbow grease and pennies on the dollar, I had a globally scalable and fault-tolerant solution. I didn’t have to manage an elastic load balancer and multiple AWS EC2 instances across Availability Zones and pay for them by the hour, whether they were idle or not. We will see how to deploy a SPA to the edge of the cloud in the *Dissecting micro frontends* section.

I was sold. I had a powerful incentive to make the change. But the story wasn’t complete.

Build-time versus runtime rendering

It’s no surprise, but no sooner did we have SPAs rendering on the client side than the JavaScript community realized we also needed rendering on the server side as well. We needed isomorphic rendering that could render on the client side or the server side. Part of a screen could be rendered on the server side and the rest could be rendered later on the client side. The part that is rendered on the server side could re-render on the client side.

So, we would need to manage servers for the presentation tier after all. Or would we? Because other elements in the current context were continuing to change as well. Continuous deployment and GitOps practices were emerging. Our ability to observe the inner workings of the presentation tier was also improving and yielding insights that would drive us in a better direction.

As I managed and tuned the older server-side rendering applications I mentioned previously, I began to realize how inefficient they were. We had designed them to be extremely dynamic. The presentation tier logic allowed almost anything to change based on the current user’s preferences, roles, and history. I watched the observability metrics roll in and it became clear that the vast majority of requests produced the same output. The requests came in and hit the servers, which hit the database and rendered the same results over and over again.

It made me think of the saying about the definition of insanity: doing the same thing over and over again and expecting a different result. This was a normal design pattern, but it was rooted in an era of long lead times and infrequent deployments. Now, the context has changed. We can deploy changes at any time, many times a day. We don’t need to make an application more complex and less efficient to accommodate change with data-driven logic. We can make a change, build it, and deploy it. This is

when **Jamstack** (https://jamstack.org) emerged, along with tools such as **Gatsby** (https://www.gatsbyjs.com) and **Next.js** (https://nextjs.org).

The **Jam** in Jamstack stands for **JavaScript, API, and Markup**. It builds on the principles of **GitOps** and continuous deployment practices that we cover in *Chapter 11*, *Choreographing Deployment and Delivery*, to create a lightweight content management system. It also leverages the power of server-side rendering at build time to deploy pre-rendered websites and SPAs that are super scalable, efficient, and cost-effective. We execute JavaScript on the client side to animate the rendering of any screen sections that need to change dynamically. We make asynchronous calls to retrieve data that changes frequently. Also, when relatively static content does change, such as a news post, we use the **Markdown** markup language to easily author the new content. Then our CI/CD pipelines build and deploy the change.

We can take this a step further and use our event-first architecture to react to business events and automate the rendering and deployment of a portion of an application, such as a micro application. With this approach, we can maximize efficiency by rendering content once, instead of over and over again. The application appears to be dynamic because it is. But we only render the content when it changes.

But there is more. Pre-rendering and deploying SPAs to the edge of the cloud has a positive impact on increasingly mobile and often disconnected users. This, in turn, has changed the equation of the web versus mobile decision.

Web versus mobile

Mobile devices and mobile applications have had a significant impact on end user expectations. Users expect to have access to their applications any time of day, from any location, with zero downtime. They understand that they may experience periods of poor connectivity, but they expect their applications to continue to function. These expectations have driven significant technical innovation. We will see some of this in the *Designing for offline-first* section.

There is technical debate over how to implement mobile applications. Should we write them using the native libraries of a specific device? Should we write responsive and **Progressive Web Applications** (**PWAs**) (https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps)? Or should we follow a hybrid approach? This is yet another framework war. However, the decision has a definite impact on the amount of work needed to support many different device types.

Building truly native mobile applications means we must implement and maintain separate applications for Android and iOS. A hybrid approach only requires a single code base but may not provide users with the native feel that they want. For a consumer-facing application, it is important to have a presence in the app stores, but for enterprise applications the opposite is true. Meanwhile, as we will see in this chapter, SPAs have evolved into mobile-ready applications.

From a business value perspective, our job as architects is clear. We need to put a solution in place that enables us to shorten lead times and gather feedback from users as quickly as possible so that we can iterate to the best solution. Once we prove that an application has business value, then we can change the implementation if there are clear improvements.

Innovation is our goal. The presentation tier is the battleground where innovation plays out. We do not want to impede innovation by placing undue restrictions on the presentation tier because we cannot easily predict how users will interact with systems in the future. For example, voice and virtual reality add new dimensions to the equation as they mature and enter the mainstream.

But we do need to tame the presentation tier so that it does not impede itself. We need the ability to change the user experience, quickly and easily, as we learn and improve our understanding of what users need in their applications. So, we need an architecture for the frontend that enables change. Let's start by looking at ways to break up the frontend.

Breaking up the frontend monolith

Our goal is to create a software architecture that enables change. In *Chapter 2, Defining Boundaries and Letting Go*, we defined the architecture of a system by dividing it into a set of autonomous subsystems. In the remaining chapters, we dive into the details of decomposing subsystems into autonomous services. But first, we need to address the presentation layer.

We need to break up the frontend monolith, just like the backend, to eliminate the friction that impedes innovation. We must decompose all the layers of the technical architecture to the same level of granularity. The frontend, the backend, and the database all need to work together as a cohesive unit. Then we can give autonomous teams control over a slice of the full stack so that they can move forward independently.

Before we get into the technical details, let’s look at how we can functionally decompose the frontend into independent but seamless applications. This will allow us to easily extend the system, top to bottom, by adding new applications without modifying existing applications.

By subsystem

In *Chapter 2*, *Defining Boundaries and Letting Go*, we aligned the autonomous subsystem architecture with the actors, business units, business capabilities, and data life cycle of the system. For example, our *Food Delivery system* has separate subsystems for customers, drivers, and restaurants. Each subsystem caters to the needs of these different actors. There are similarities across these subsystems, such as all of them having a sign-up process, but the details vary significantly.

Each subsystem should maintain its frontend applications. A subsystem’s frontend will have a main entry point for its users that provides access to all its features. These features will be separately deployable frontend applications.

By user activity

Within a subsystem, we decompose the frontend into multiple applications along the lines of the activities the users perform. We call these micro applications or micro-apps. For example, the frontend for the customer subsystem of our Food Delivery system contains the following micro applications:

- The main entry point is a **Jamstack** application that provides static marketing material, sign-in and sign-up links, and navigation to start

browsing restaurants and menus.

- Sign-in and sign-up are separate but related applications.
- Searching for and browsing restaurants and menus is another application.
- Adding items to the cart and the cart itself are separate micro applications.
- Checkout is another micro application and tracking the status of an order is another application.
- Managing a user's account is another set of micro applications for profile settings, order history, and so forth.

Each of these micro applications is separately deployable and likely owned by separate autonomous teams. We will see how to seamlessly stitch these applications together in the *Dissecting micro frontends* section. Each micro application has its own BFF service and database, which we cover in *Chapter 6, A Best Friend for the Frontend,* and in *Chapter 5*, *Turning the Cloud into the Database*. This last point is significant because it enables autonomous teams by giving them control over the full stack.

By device type

Today's users are on the move and interact with a system using multiple devices and device types. For example, you might order food from your phone when you are on your way home and then you track the order from a laptop or a tablet once you get home. Handling all the device type permutations can impact our ability to change a system quickly and easily.

Depending on the system, it may not make sense to support all activities on all device types. For example, a data entry-intensive activity may not be practical on a small phone. Or we might support the ability to perform part

of an activity from a mobile device when the user is on the move and then complete the activity when the user is on a laptop. Some activities may only be available based on the type of device.

Up to a point, we can make an application responsive based on the type of device and automatically adjust the layout to accommodate the screen size. However, there are limits, and implementing a responsive screen may be counter-productive in some scenarios. In these cases, it may make sense to create separate applications for different device types. Then we direct the user to the right application based on the current device type.

By version

In *Chapter 11*, *Choreographing Deployment and Delivery*, we discuss a versioning strategy to achieve zero-downtime deployments using the Robustness principle and feature flags. When we are making small, incremental changes to an application, we do not need to support multiple versions. However, feature flags can get messy, when we are making significant changes. In these cases, it is better to create a new version of an application and run both versions in parallel until the new version is complete. We use a special role as the single feature flag to limit access to the new version of the application.

Now, let’s see how we can stitch all these micro applications together to create a seamless user experience.

Dissecting micro frontends

To achieve our goal of creating a software architecture that enables change, we need to give teams full control of the entire stack. As depicted in *Figure*

3.1, a single team should own a micro application, its BFF service, and the datastore. This reduces the need for inter-team communication, which would add dependencies on another team's schedule:

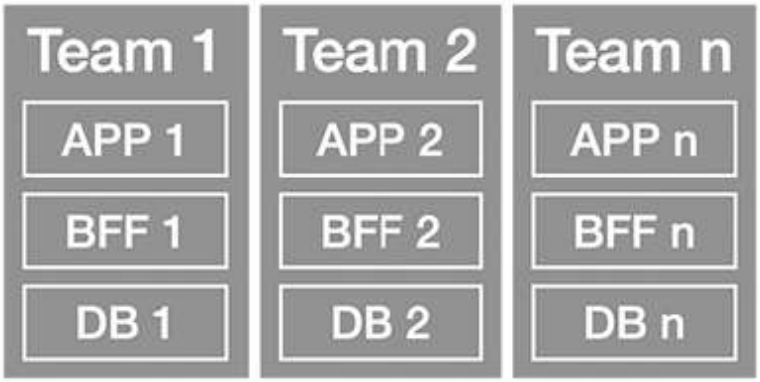


Figure 3.1: Full-stack teams

These self-sufficient, full-stack, autonomous teams are able to move at their own pace, minimize lead time, and respond to user feedback at will, precisely because they are not dependent on other teams. In *Chapter 6, A Best Friend for the Frontend*, we will see how teams implement BFF services, and in *Chapter 5, Turning the Cloud into the Database*, we will see how to decompose monolithic databases.

In this section, we look at how to create a micro frontend. The objective of a micro frontend is to divide the user experience into a set of independent micro applications, while also providing users with a seamless experience. As users move between different activities, it should not feel like they are jumping between applications. It should feel like a single application. As depicted in *Figure 3.2*, we may compose a single screen from many micro applications, but they will appear as a single application to the user:

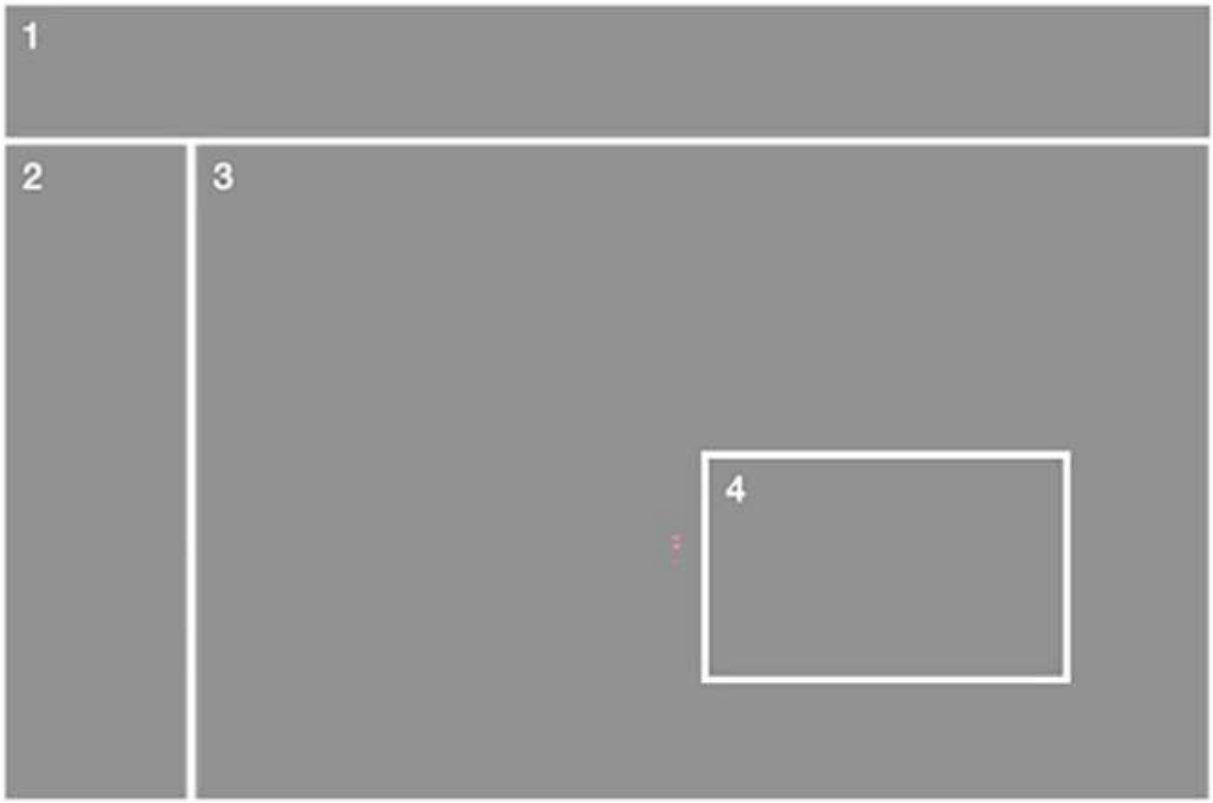


Figure 3.2: Micro frontend

The preceding illustration shows the banner (**1**) and left navigation (**2**) as separate micro applications. Another micro application (**3**) provides the contents of the page, which contains yet another micro application (**4**). As the user navigates through the subsystem, different micro applications provide the content of the different screens. But from the user's perspective, it is all one application.

The micro frontends approach provides the following three major benefits:

- **Isolates change**: Each micro application is independently deployable. Following the Open-Closed principle, we can deploy a new micro-app without changing existing applications. Following the Single Responsibility principle, each micro-app is responsible to a single actor, who drives change. For when we do need to change and redeploy an

existing micro-app, we have limited the scope of the impact to that one micro-app. In other words, micro applications are autonomous, and they enable autonomous teams.

- Maximizes the potential for lazy loading: Monolithic frontends are notorious for long load times and the problem grows more acute with each new feature added. Code splitting provides a workable solution to this problem, but micro-apps bring lazy loading to the forefront, instead of being an afterthought. An individual micro-app can and will leverage code splitting as well. This level of control is very important for mobile users.
- Supports concurrent use of multiple frameworks and, more importantly, multiple framework versions: Each micro-app can use a different framework, such as React or Angular. However, we should not abuse this capability. Each subsystem should standardize on a single framework. We should only utilize multiple frameworks as a migration path to the standard framework. This capability also enables the upgrading of a single framework, without requiring all micro applications to upgrade at once. This is significant, as there is constant innovation in these frameworks. New micro-apps can use the latest version and older apps can upgrade as they change.

There are many alternatives for implementing micro frontends. We can assemble them at build time or runtime, on the client side or the server side; we can link them together or we can programmatically connect them. We will look at an example throughout this chapter that utilizes the **single-spa** (`https://single-spa.js.org`) framework to programmatically stitch the micro frontend together on the client side at runtime. There is a **main** application that bootstraps the system, and each micro-app implements

a well-defined life cycle so that the system can activate them and mount them at well-defined locations on the screen.

You can find a micro frontend example here: https://github.com/jgilbert01/micro-frontend-template. The *(post build)* code snippets in the remaining sections will highlight some of the core aspects.

Let's dig into the details of creating a micro frontend.

The main app

The presentation layer of each autonomous subsystem has a main entry point for gaining access to all of its features. However, the **main** application does not provide any specific features, other than bootstrapping the micro frontend environment. In essence, the **main** application just owns the `index.html` file.

The main application has its own source code repository and its own CI/CD pipeline. The CI/CD pipeline performs the following duties:

- It creates the object store, such as an S3 bucket, that will hold all the resources of the various micro applications.
- It creates the Content Delivery Network (CDN), such as an AWS CloudFront distribution, that will serve all the resources of the various micro applications from the edge of the cloud.
- It uploads the `index.html` file and a manifest file to the bucket. We will cover manifest files in the *Manifest deployer* section.
- It creates the domain name that we use to access the `index.html` file, such as with an AWS Route 53 record set.

The bootstrapping process starts when we access the `index.html` file.

The index file

The `index.html` file is lean and trim. It initializes the dynamic module loading system and kicks off the micro application registration process. This is how it appears:

```
<html> <head> <title>My Main</title> <meta name="importmap-type"
content="systemjs-importmap"/> <script type="systemjs-importmap"
src="/importmap.json"/> <script src="https://cdn.../system.min.js"
/> <script src="https://cdn.../amd.min.js" />
<script>System.import('@my/main');</script> <link ...
href="//cdn.../semantic.min.css" /> </head> <body /> </html>
```

The primary job of `index.html` is to load the `/importmap.json` data structure and initialize the SystemJS (https://github.com/systemjs/systemjs) module loader. Then it imports the `@my/main` micro application to kick off the registration process. Finally, it loads the global style sheet.

The importmap file

Lazy loading is one of the primary benefits of the micro frontend approach. We load each micro application just in time. This provides a more responsive user experience since we do not spend time loading applications that we do not use.

However, by default, JavaScript does not support dynamic module loading. Take the following `import` statements as an example:

```
import React from 'react'; import { useMountPoint } from
'@my/shared';
```

Unless we specify otherwise, a bundler, such as **webpack**, will pull these dependencies into the bundle that it creates. This increases the size of the

bundle and duplicates these dependencies across multiple micro applications.

Alternatively, we can tell the bundler to externalize specific dependencies. Now we need a mechanism to dynamically load these modules when our code imports them. This is where the `importmap.json` file and `SystemJS` come into play. Here is an example of an `importmap` file:

```
{ "imports": { "@my/main": "/my-main/{commit-hash}/my-main.js",
"@my/main/": "/my-main/{commit-hash}/", "@my/app1": "/my-
app1/{commit-hash}/my-app1.js", "@my/app1/": "/my-app1/{commit-
hash}/", "@my/shared": "/my-shared/{commit-hash}/my-shared.js",
"@my/shared/": "/my-shared/{commit-hash}/", "react":
"https://cdn.jsdelivr.net/...", "react-dom":
"https://cdn.jsdelivr.net/...", "semantic-ui":
"https://cdn.jsdelivr.net/...", "single-spa":
"https://cdn.jsdelivr.net/...", } }
```

This file does what its name implies. It maps the import names of modules to the path where they are located. SystemJS uses this file to import externalized modules. When we import the `@my/main` module, SystemJS will load it from our S3 bucket.

Note that there are two entries for each of our micro applications. We use the first entry to bootstrap a micro application without loading the entire module. Then we use the second entry when it is time to load a micro application.

We will cover the life cycle of a micro application in the *Micro-app activation* section. But first, we need to register the applications.

Micro-app registration

We need to register each micro application with the `main` application. The registration process begins when we import the `main` application at the

bottom of the `index.html` file. The following code block from the main application shows the registration process:

```
import { registerApplication, start } from "single-spa"; ...
fetch('/apps.json').then(res => res.json()) .then((apps) => {
const routes = constructRoutes(apps); const applications =
constructApplications({ routes, loadApp: ({ name }) =>
System.import(name), }); const engine = constructLayoutEngine({
routes, applications });
applications.forEach(registerApplication); engine.activate();
start(); });
```

The main application retrieves and processes the `apps.json` manifest file. It constructs the routes and registers all the applications, and then it calls `start` to activate the system. It also defines a `loadApp` function for each application. This function will lazily load the entry file for a micro application. This is the first entry for the application in the `importmap` file. We will cover the entry file shortly, in the *Micro-app* section. Here is an example of an `apps.json` file:

```
{ "routes": [ { "type": "application", "name": "@my/nav" }, {
"type": "route", "path": "/things", "routes": [{ "type":
"application", "name": "@my/app1" }] }, ... ]}
```

The single-spa layout engine defines the structure of the `apps.json` file. The file contains all the applications and their activation policies. For example, the `@my/app1` micro application will activate when a user accesses the `/things` route and the `@my/nav` micro application is always active. We will see how we use this information in the *Micro-app activation* section. We will cover the creation of this file in the *Manifest deployer* section.

Now, let’s look at the resources of a micro-app.

Micro-app

In the *Breaking up the frontend monolith* section, we discussed various ways of decomposing the frontend into a set of micro applications. We also just covered dynamic module loading and application registration. Now let's look at what goes into a micro-app.

Each application has its own source code repository and its own CI/CD pipeline. The CI/CD pipeline performs the following duties:

- It tests all of the micro application's source code. We will cover testing in *Chapter 11*, *Choreographing Deployment and Delivery*.
- It builds the micro application's bundle with a bundler, such as webpack.
- It uploads the bundle and a manifest file to the bucket created by the main application. We will cover manifest files in the *Manifest deployer* section.

The internals of a micro application are the same as for your chosen framework, such as React. The only difference is the entry point. A micro application has an entry file and a root component. Let's look at each in turn.

The entry file

Each micro application must have an entry file. We externalize this file from the rest of the micro application's bundle to maximize lazy loading. The main application will lazily load this file using the `loadApp` function that we defined in the *Micro-app registration* section. The following code block shows the core elements of the entry file:

```
import singleSpaReact from 'single-spa-react'; ... const
lifecycles = singleSpaReact({ ... loadRootComponent: () =>
import('./root.component.js'), domElementGetter: () => document
.getElementById('container'), ... }); export const bootstrap =
```

```
lifecycles.bootstrap; export const mount = lifecycles.mount;
export const unmount = lifecycles.unmount;
```

The entry file defines the following micro application life cycle functions:

- The `loadRootComponent` function lazily loads the micro application's bundle.
- The `bootstrap` function is called once to resolve the root component.
- The `domElementGetter` function returns the element where we want the root component mounted.
- The `mount` function renders the application in the location defined by `domElementGetter`.
- The `unmount` function removes the DOM elements that we rendered during mounting.
- We will see how we use these life cycle functions in the *Micro-app activation* section.

Root component

- The actual functionality of a micro application starts in the root component. This is analogous to the App component in a typical React application, such as the App component generated by the Create React App utility. The following code block shows a basic root component:

```
import React from "react"; const Root = (props) => (<p>
{props.name} is mounted!</p>); export default Root;
```

- This root component example is just the equivalent of hello world for micro applications. It displays the name of the micro application. The typical React root component will initialize various context providers and set up routing rules. We will see a more elaborate example of this in the following *Micro-app activation* section.

Micro-app activation

For a typical monolithic SPA, the browser will load the entire code base when the user accesses the page. This process grows more and more time-consuming as the monolith grows larger. The micro frontend approach solves this problem by dividing the application into a set of lazily loaded micro applications. Then we load and activate a micro-app just in time.

Let’s bring all this metadata and code together and see how we activate micro applications. We typically activate micro applications based on routes and we can manually activate them as well. But first, let’s look at the full life cycle flow.

Micro-app life cycle

As we saw in the *Entry file* section, each micro application provides a set of life cycle methods that the micro frontend framework uses to bootstrap, mount, and unmount the application. The following steps outline the micro-app life cycle:

1. At startup, the main application registers each application in the `apps.json` manifest and starts the micro frontend framework.
2. Then the framework checks the activation policy of each application.
3. The first time an application activates, the framework loads the entry file to gain access to the application’s life cycle functions. Then it calls the bootstrap life cycle function to load the root component.
4. Each time an application activates, the framework calls the `mount` life cycle function to attach the root component to the DOM as specified by the `domElementGetter` function.
5. When an application deactivates, the framework calls the `unmount` life cycle function to remove it from the DOM.

6. As the user navigates the systems, the framework checks all the activation policies and mounts and unmounts applications accordingly.

Some micro applications are always active. For example, a global navigation micro application may always be available to provide a main menu feature. Most micro applications will activate when the user selects these menus. Also, to a lesser degree, one micro application will manually activate another.

Route-based activation

In a SPA, routing is the practice of rendering different components as the URL changes. SPA frameworks hook into the browser history and location interfaces and render the appropriate components as the user navigates through the application. These routes represent different user activities. In the *Breaking up the frontend monolith* section, we identified that it is a best practice to create a separate micro application per user activity.

Routes are the most common way to activate micro applications. The micro frontend framework uses top-level routes to identify when to activate a specific micro application. Then the micro application uses sub-level routes to direct the rendering of its components. The following code block shows the `Root` component of a micro application that defines sub-routes:

```
const Root = (props) => ( <Router> <Things path="things"
{...props} /> <Thing path="things/:id" {...props} /> </Router> );
```

The micro frontend framework will activate this micro application based on the /things route, as defined in the apps.json file. Within this micro application, the /things route will render the Things component to browse things and the /things/:id route will render the Thing component to view a specific thing.

Now let’s see how we can manually activate a micro application.

Manual activation

It is also possible to take full control and manually activate a micro application. We will typically do this when we want to embed one micro application within another micro application. For example, in our Food Delivery system, we embed the add-to-cart micro application inside the view-menu-item micro application.

The following code block shows manually activating another micro application:

```
<div> <Parcel config={() => System.import('@my/add-to-cart')}
{...props, additionalProp } /> </div>
```

The `single-spa-react` library provides the `Parcel` component. We place this component in the location where we want to mount the micro application. This means that embedded micro applications do not need to declare the `domElementGetter` function. Then we configure a `load` function and specify the name of the micro application we want to load. We can also pass properties to the embedded micro application to initialize it with the proper information.

Now, let’s see how we can dynamically associate micro applications to mount points.

Mount points

A micro application mounts to a specific `dom` element when it activates. We refer to this `dom` element as a mount point. A mount point represents a contract between the owner of the mount point and the micro applications

that use the mount point. For example, the owner controls the location, width, and height of the `dom` element, and we must design a micro application to work within those parameters.

Navigation is a typical example of a mount point. For example, as shown in the following screenshot, the left navigation menu will define links to activate micro applications based on routes and define a container `dom` element where the applications will mount. As another example, the top-right navigation may open a sliding drawer and manually activate the specified micro application. In either case, we design the micro applications to make appropriate use of the given screen real estate.

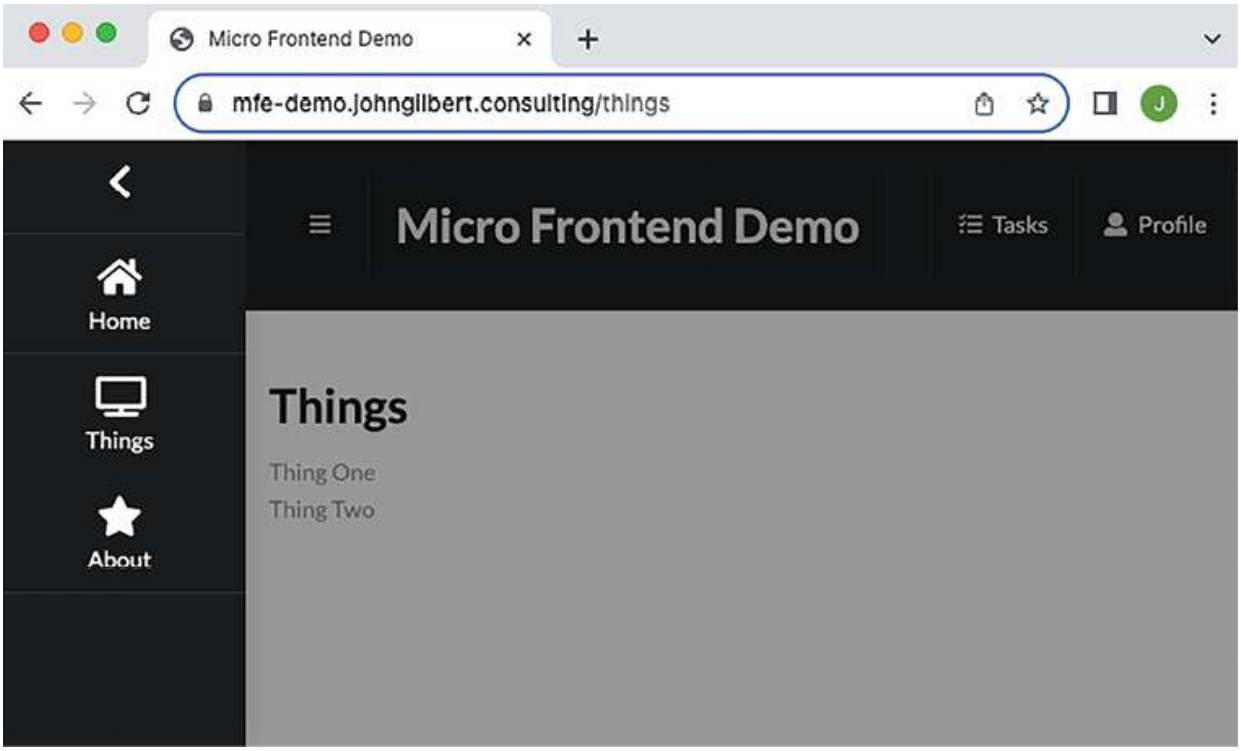


Figure 3.3: Mount point demo

For some systems, it may be reasonable to hardcode the navigation for a fixed set of micro applications. However, for a large application, we prefer to dynamically generate the navigation based on metadata. This allows us to

add and remove micro applications without making changes to the navigation source code. The following code block shows an example of a dynamically rendered navigation component:

```
export const LeftNavMenuItems = () => { const { items } =
useMountPoint('left-nav'); return items.map((item) => <Menu.Item
key={item.key} content={item.content} icon={item.icon} as={Link}
to={item.to} />); }; export const MainContainer = ({ children })
=> <div id='container'>{children}</div>;
```

The `useMountPoint` Hook retrieves a `/mount-points.json` file and returns the requested metadata. This component loops over the metadata and renders menus with links that will route to the micro applications. We also define the container element where we intend to mount the micro applications. Here is an example of a `mount-points.json` file that provides the metadata:

```
"mount-points": { "left-nav": [{ "key": "thing", "content":
"Things", "icon": "tv", "to": "/things" }, ...] }
```

The mount point code dictates the contents of the metadata for a specific mount point. In this example, the mount point expects a unique key, the content and icon to display, and the route that the link will navigate to. When a user clicks on a menu link, the framework will then activate a micro application following the process that we outlined in the *Micro-app activation* section.

We can take mount points further and add conditional rules to further control navigation. For example, we may filter links based on the user's role or device type. We may disable menus based on the state of the domain entities. We would add these criteria to the metadata, such as a list of valid roles, user agents, or status flags. We will see an example of this in *Chapter 10, Securing Autonomous Subsystems in Depth*.

Now let’s see how manifest deployer automates the process of bringing together all the metadata from our various manifest files.

Manifest deployer

As we have seen in the preceding section, we have three manifest files that allow us to dynamically create a micro frontend from a set of micro applications. These are the `importmap.json`, `apps.json`, and `mount-points.json` files. These files and the micro frontend framework allow us to independently deploy micro applications without modification to the other micro applications.

However, we do not want to risk manually updating these master manifest files as we add and change micro applications. Instead, each application defines its own manifest file that contains the fragments that we use to generate the main manifest files. Here is an example of an `mfe.json` manifest file:

```
{ "orgName": "my", "projectName": "app1", "importmap": { ... },
"apps": { ... }, "mount-points": { ... } }
```

Each file must have `orgName` and `projectName` fields that uniquely identify the micro application, plus `importmap` and `apps` elements. The `mount-points` element is optional. The CI/CD pipeline for a micro application uploads its `mfe.json` manifest file to the S3 bucket. This in turn triggers the manifest-deployer service. This service builds the master manifest files from all the individual manifest files and stores them in the S3 bucket. The deployer service leverages the same stream-processing framework we cover in *Chapter 4, Trusting Facts and Eventual Consistency*.

Now let’s see how micro applications can communicate with each other.

Inter-application communication

Now that we have divided the system into micro applications, we need mechanisms that allow the applications to communicate with each other. For example, one application may launch another, or multiple applications on a screen may need to update together as the context changes.

In the *Breaking up the frontend monolith* section, we identified several best practices we can use to decompose a system into micro applications. We minimize the need for inter-application communication when we follow these best practices because each user activity is self-contained.

If we find that two micro applications need to talk back and forth, then this may be an indication that we should merge them. But there are plenty of non-cyclic scenarios where one application interacts with another application.

One application can pass information into another application using route parameters. For example, one application may solely provide a search feature. From the search results list, we may want to launch various user activities. We would implement each of these user activities as a separate micro application. Then we pass identifier information as a parameter, such as the selected product ID, so that the micro application can retrieve the appropriate information. The same is true for manually activating a micro application and passing properties.

We can also create micro applications that act as a shared library. They export functions and we import them dynamically using `importmap`. This ensures that only a single instance of the library exists. This allows us to cache common data within these modules and access it through the exported functions.

We can also use an event bus to exchange information with minimal coupling. For example, one application can register an event listener using `window.addEventListener('my-evt', ...)`. Then, other applications can send out information as their state changes using `window.dispatchEvent(...).`

It is also possible to share state information across micro applications with libraries such as Redux, but this limits the independence of the micro applications. Each micro application should maintain its own state store. We should limit any shared or global storage to a very specific scope that lives in a shared micro application with a clearly defined and exported interface.

The micro frontend approach is a significant architectural improvement that brings the flexibility of independent deployments to the presentation layer. We have divided the system into multiple micro applications so that they can all change independently. This enables autonomous teams that have control over the full stack, so they can reduce lead times.

Now, let’s take a deeper look at state management and how to design our micro applications to work offline, so that mobile users can work when disconnected.

Designing for offline-first

Today’s users are on the move, and they increasingly rely on mobile devices. You may be reading this on a mobile device. If so, you likely downloaded the content to your device, so you could read it when you are offline. For static content, this is straightforward, and you knew upfront that you wanted offline access.

For applications with dynamic content, the process of making the content available offline is more involved. Plus, it is more difficult to predict when connectivity will be unavailable. It is helpful to think of this problem in terms of the CAP theorem, which shows us that in the case of a system partition, we have to choose between consistency and availability. We cannot have both.

Loss of connectivity is a perfect example of a system partition because the application can no longer retrieve and update the dynamic content. In this case, users favor availability over consistency. They want the application to continue working and for it to sort things out when the connection is reestablished. This means that mobile users are driving the need for eventually consistent systems.

We discuss eventual consistency in detail in *Chapter 4, Trusting Facts and Eventual Consistency*, and we cover related data patterns in *Chapter 5, Turning the Cloud into the Database*. In this section, we cover the concept of session consistency and extend it to support disconnected users. In other words, we will design the presentation tier to work offline-first. It will cache data locally and be fully functional when disconnected.

Let’s start by looking at how an application can be transparent with the user about the current status of the system. Then we will cover the many facets of caching on the frontend. Finally, we will see how we can keep the frontend in sync with the backend.

Transparency

Transparency is the first order of business for offline-first applications. We need to provide users with visibility of the status of the application so that they can make informed decisions. This can range from very basic indicators

to more elaborate reconciliation features, based on the needs of the application and the experience of the users. A typical mobile email application provides an excellent frame of reference.

Status indicators

First and foremost, an application needs to provide the user with some sort of connectivity status indicator. This is in addition to the indicator provided by the device, because an application may only lose connectivity to its backend, such as during an outage. For example, an email application will indicate when it cannot reach a specific email account.

Next, it may be helpful to provide information about the cache, such as the version of the application code and the time of the most recent data synchronization. In some cases, it may be useful to provide an as-of date (that is, a timestamp) for each record. The indicator should also show when synchronization is in progress. Again, an email app indicates when an account was last synchronized and when a sync is in progress, and, of course, the individual messages have timestamps.

Outbox

Things get more interesting when the user edits dynamic content offline. We need to queue the changes and push them to the backend when we have a connection. We need to make this clear to the users. They need to understand that their work is incomplete, and they may need to intervene in the worst case.

An outbox metaphor, such as in an email application, may be a good option. An application may already maintain status fields on the data, and this could be an additional intermediate status.

Inbox

Once we have a connection and synchronization is complete, there may be certain updates that are important to highlight for the user. For example, there may be changes that are related to the data that the user had in the outbox. This may just be informative, but in the worst case, the user may need to reconcile changes if there were any conflicts with changes performed by other users. In this case, it may be helpful to keep a visible list of all the changes we sent as well.

It is important to give users control. They can initiate a cache synchronization to increase their confidence in the data. Or they can initiate a synchronization when they know they are going offline. Let’s see how the caching mechanism works.

Local cache

An offline-first application maintains a local cache on the client device so that the application remains useable when there is limited to no connectivity. We need to cache the code of the micro apps, we need to cache the latest data, and we need to be ready to cache our writes. Let’s take these one at a time and see how they build on each other.

Caching code

Traditionally, we install an application by downloading it from an app store and allow it to automatically update when changes are available. As we covered in the *Web versus mobile* section, this approach comes with some choices and dependencies we may wish to avoid.

The web, on the other hand, traditionally downloaded resources just in time, which meant that a web application was only available when online. Cache-control headers allow a browser to cache content to improve performance and minimize data transfer, but a browser will still produce errors on cached content when there is no connectivity.

A PWA is a newer alternative that strives to provide the benefits of both these approaches. A PWA is a normal web application that runs in the browser, but it progressively downloads and caches the application code and eventually runs the entire application from the cached code as though we installed it all initially.

We turn our micro frontend into a PWA by adding a manifest file and a `ServiceWorker` to the main micro app. All the other micro apps will automatically benefit. Here is an example of a `manifest.json` file:

```
{ "short_name": "My Apps", "name": "My Apps", "background_color":
"#000000", "display": "standalone", "start_url": ".", "icons": [{
"src": "/public/logo.png", "type": "image/png", "sizes": "144x144"
}] }
```

A link to the manifest file in the `index.html` file tells the browser to enable certain features, such as the ability to install the application as an icon on the desktop or home screen. The file contains additional metadata, such as the application name and the icon. Note that browsers can be finicky about these settings. For example, the specified icon size must match the actual size of the specified file, otherwise these features will not be enabled.

Next, we need to register a `ServiceWorker`. Here is an example:

```
if ('serviceWorker' in navigator) { navigator.serviceWorker
.register('/service-worker.js'); }
```

This code lives in the entry file of the main app. First, we check to see if the browser supports service workers. In other words, we will progressively add these features to our micro frontend to help ensure it can run on the widest array of browsers. Then we register the path to the `service-worker.js` file.

The `ServiceWorker` instance runs in a separate thread and acts as a proxy for network requests. It intercepts requests, detects whether there is connectivity, and gives the application an opportunity to utilize the browser's Cache API (https://developer.mozilla.org/en-US/docs/Web/API/Cache), which is distinct from the browser's HTTP cache. Before we dive into the code, let's cover the caching strategies we will use:

- The **Cache-First** strategy always checks the cache first. If the resource is not in the cache, we fall back to the network and store the response in the cache for next time. We will use this strategy for resources that change infrequently.
- The **Network-First** strategy always checks the network first. It stores the latest response in the cache to use as a fallback when we do not have connectivity. We will use this strategy for resources that change more frequently. Note, when the `ServiceWorker` makes these network calls, it will observe the browser's HTTP cache, which is driven by any cache-control headers.
- The **Stale-While-Revalidate** strategy is a middle ground. It uses the resources in the cache but refreshes them from the network in the background for next time. We will use this strategy when we want to prioritize speed over freshness.

Now we need to implement the `service worker`. This code can be tedious, and it is largely boilerplate, so we will use a set of recipes and utilities from

the Workbox library (https://developer.chrome.com/docs/workbox) to simplify the code. Here is an example of a service worker:

```
import {...} from 'workbox-recipes'; import {registerRoute} from 'workbox-routing'; import {NetworkFirst} from 'workbox-strategies'; // cache first recipes imageCache(); googleFontsCache(); // stale while revalidate recipes staticResourceCache(); // network first recipes pageCache(); registerRoute( /.*\.json/, new NetworkFirst({ cacheName: 'json', }), ); // catch recipe offlineFallback({ pageFallback: 'index.html' });
```

This example uses multiple recipes and cache strategies. The image and font recipes use the cache-first strategy. The static resources recipe uses the stale-while-revalidate strategy for scripts and styles. The page recipe uses the network-first strategy for page navigations. We roll our own network-first cache for our `importmap.json`, `apps.json`, and `mount-points.json` files. And finally, when all else fails, we send users to the index page.

Once a user accesses the home page, we will have cached the skeleton of the application, such as the header, footer, sidebars, navigation, and landing content, so we will always have something to fall back to. As the user moves through the system, we will cache more and more of the micro apps, so that the user can access more functionality when offline.

There are many advanced options we can leverage to cache the apps more aggressively. For example, we can have webpack set up a precache and we can utilize the information in our `importmap.json`, `apps.json`, and `mount-points.json` files to warm the cache with one fell swoop based on the user's permissions and the user agent of the current device. But the example above is a great place to start, and you can optimize the service worker as you learn the application usage patterns.

Now let's see how we can use the service worker to implement a read cache for our dynamic data.

Caching data reads

So far, we have installed the application code, but it doesn't do much without data. To gain access to data, we need to make remote calls to BFF services, which we cover in *Chapter 6, A Best Friend for the Frontend*. But we won't have access to these services when we are offline, so we need to maintain a local cache that stores data on the user's device.

We will use a combination of the `Network-First` and `Stale-While-Revalidate` caching strategies for our data since it changes dynamically. The following code block shows an example of setting up a data cache in our service worker:

```
import {NetworkFirst} from 'workbox-strategies'; import
{ExpirationPlugin} from 'workbox-expiration'; registerRoute( ({
request }) => request.url.includes('/api-')), new NetworkFirst({
cacheName: 'api-gets', plugins: [new ExpirationPlugin({
maxAgeSeconds: 45 * 24 * 60 * 60, // days })], }), 'GET' );
```

In this example, we set up a `NetworkFirst` cache for all `GET` requests to our BFF services. In *Chapter 9, Running in Multiple Regions*, we will see how we configure all our BFF service endpoints with the `/api-` prefix. We also configure this cache to expire after 45 days, so that it does not grow unbounded. This basic example will cache any data that the user is currently working with.

To be fully prepared for offline mode, we also need to proactively fetch certain types of data. For example, we need to cache reference data that the UI needs to populate drop-down lists. In *Chapter 6, A Best Friend for the Frontend*, we will introduce a *List-of-Values* service that facilitates this

cache. We can preload this data on application startup and use a `Stale-While-Revalidate` strategy, so that we refresh this data in the background.

We may also want different expiration periods for different services. Plus, we can combine this with the use of **cache-control** headers in the individual service responses to fine-tune the behavior and leverage the browser's HTTP cache as well. Furthermore, we can use libraries such as `react-query` (https://www.npmjs.com/package/@tanstack/react-query) to maintain an in-memory cache at the application level to fully optimize the responsiveness of the UX.

At this point, we have a working application and data for the user to interact with when offline. Now let's see how we can implement a write cache that ensures all writes make it to their BFF service.

Caching data writes

Up to this point, we have cached the application code and pertinent data in the local cache, so that an application works offline. If the application allows users to create or modify data, then we need to make remote calls to BFF services to persist the changes. But we won't have access to these services when we are offline, so we need to cache the user's changes locally and push changes to remote services when a connection is available.

We also need to provide the user with **session consistency**. In other words, we need to guarantee read-your-writes consistency. When offline, we cannot write changes to the BFF service, so we cannot read the changes from the BFF service either. So, we will need to keep the read cache in sync with any local changes made by the user.

Let’s start by storing a draft copy of any changes the user is making. The following code block shows an example of storing/restoring form values to/from local storage:

```
import {useForm} from "react-hook-form"; import useFormPersist
from 'react-hook-form-persist' const {..., watch, setValue} =
useForm(); useFormPersist("myForm", { watch, setValue, storage:
localStorage});
```

In this example, the `useFormPersist` hook uses the `watch` function to subscribe to input changes and write the values to local storage. Then it can use the `setValue` function to restore the values from local storage. When the user submits the form, then the values are cleared from storage. This allows us to recover the user’s draft work if there is a disruption.

Next, we need to update the local cache when the user submits a change. The following code block shows an example of implementing session (that is, read-your-writes) consistency with `react-query`:

```
const qc = useQueryClient(); useMutation({ mutationFn: ...,
onSuccess: (data) => { qc.setQueryData(['order', data.id], data)
qc.setQueriesData(['orders'], (prev) => prev.map((d) => (d.id ===
data.id ? data : d))) }});
```

This is an example of an update mutation. We register an `onSuccess` function that will update the cache with the mutated data after we perform a write. First, we update the `order` details cache with the new version of the data. Then we find the previous version in the `orders` list cache and replace it with the new version. For an insert mutation, we would add the data to the orders cache and for a delete mutation, we would remove it. This will cause the UI to re-render with the modified data so that we can read our own writes.

Next, we need to implement an outbox that will queue our mutations when we are offline and execute them in the background when we regain connectivity. The following code block shows an example of implementing the background process in our service worker:

```
import {BackgroundSyncPlugin} from 'workbox-background-sync';
const bgSyncPlugin = new BackgroundSyncPlugin('outbox', {
maxRetentionTime: 24 * 60, // retry for max hours });
registerRoute( ({ request }) => request.url.includes('/api-')),
new NetworkOnly({ plugins: [bgSyncPlugin], }), 'PUT', );
```

In this example, we are using the background sync provided by the Workbox library. We add this code to our service worker. First, we initialize the `BackgroundSyncPlugin` that will retry transactions in the background for up to 24 hours. Then we register the plugin for `PUT` requests to our BFF services (that is, URLs starting with `/api-`). We will register for `DELETE` requests as well. The plugin stores the requests in `IndexedDB` (https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API). So, we can use this data to provide a view of the outbox to the user. Meanwhile, in the previous example, we have updated the cache so that users can continue to work with the data.

We will revisit this in *Chapter 9*, *Running in Multiple Regions*, where we dissect the behavior of the system during a regional disruption. Now that we have data cached locally, we need to keep it up to date with any changes. Let's see how we can utilize a live update service to keep data synchronized in near real time.

Live updates

We understand that cached data can become stale when an application goes offline. This is a compromise we are willing to make so that we can continue to work when we lose connectivity. Yet, we want to bring the cache up to date as quickly and as efficiently as possible. We can expire a cache after a time period, but this can allow the data to go stale in the meantime. We can query for updates repeatedly, but this is inefficient. We need a better solution.

Instead, we will implement a live update solution that capitalizes on our event-first architecture. In *Chapter 4, Trusting Facts and Eventual Consistency*, we cover the core concepts of this architecture, and in *Chapter 5, Turning the Cloud into the Database*, we will see how autonomous services leverage events to implement the CQRS pattern and maintain their own living cache that continuously updates as data changes. In this section, we will extend events to the client side with a live update service.

Let's look at two alternative implementations of the live update service. First, we will look at the WebSocket implementation and then the long-polling alternative.

WebSocket

Our system is composed of autonomous services. These services publish domain events to an event hub as their state changes. A micro application interacts with a specific set of domain entities and maintains them in a local cache. To keep this cache up to date, we want the frontend to subscribe to domain events, so that we can update the local cache in near real time when we are online. *Figure 3.4* depicts how we can implement this solution with WebSocket:

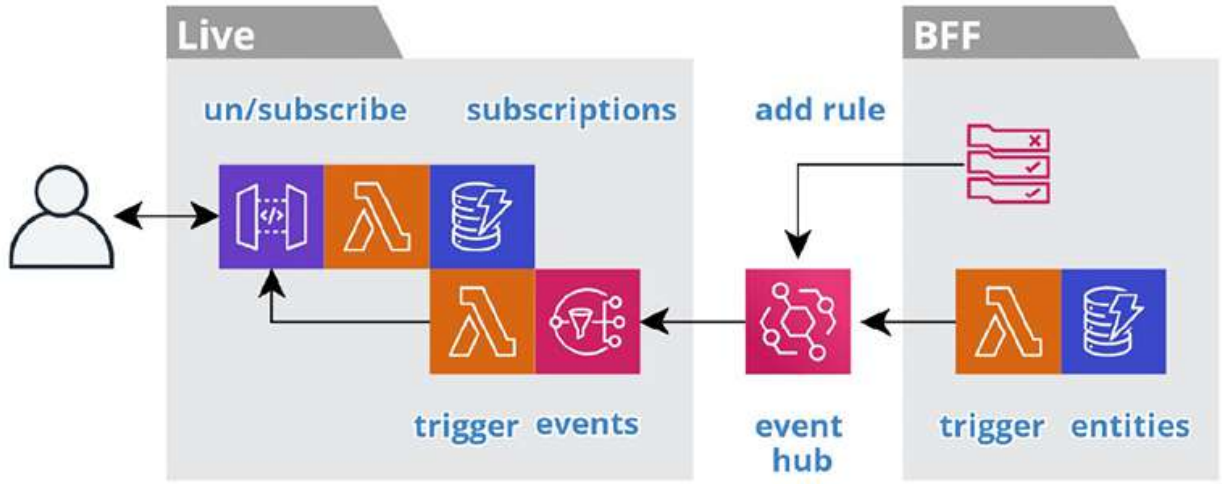


Figure 3.4: Live updates – WebSocket

The **Live** update service defines an **events** topic, using AWS SNS. The **BFF** services for the various micro applications **add rules** to the **event hub** for the specific domain events they want to route to the **events** topic. A shared micro application initiates a connection with the **Live** update service to create the WebSocket for the user and the callback URL for the WebSocket is stored in the subscriptions database. Individual micro applications can add specific subscriptions to control the events the user will receive.

A **trigger** function reacts to events as they arrive on the **events** topic. The function looks up subscriptions in the database and forwards the events to the users' callback URLs. The shared micro-app receives the events through the WebSocket on the client side and makes the events available to other micro applications through a window events listener, as we discussed in the *Inter-application communication* section.

From here, the individual micro applications take over. For example, a React-based micro application may leverage **react-query** for state management and persist the state to the local cache. The shared micro app forwards (that is, dispatches) the domain events to subscribing micro apps,

which update the state of their cache and consequently update the screen with the new values.

Long polling

The WebSocket implementation works well when an application is online but does not directly support an application when it is offline. *Figure 3.5* depicts an alternative implementation that uses AWS SQS to buffer events when a user is offline:

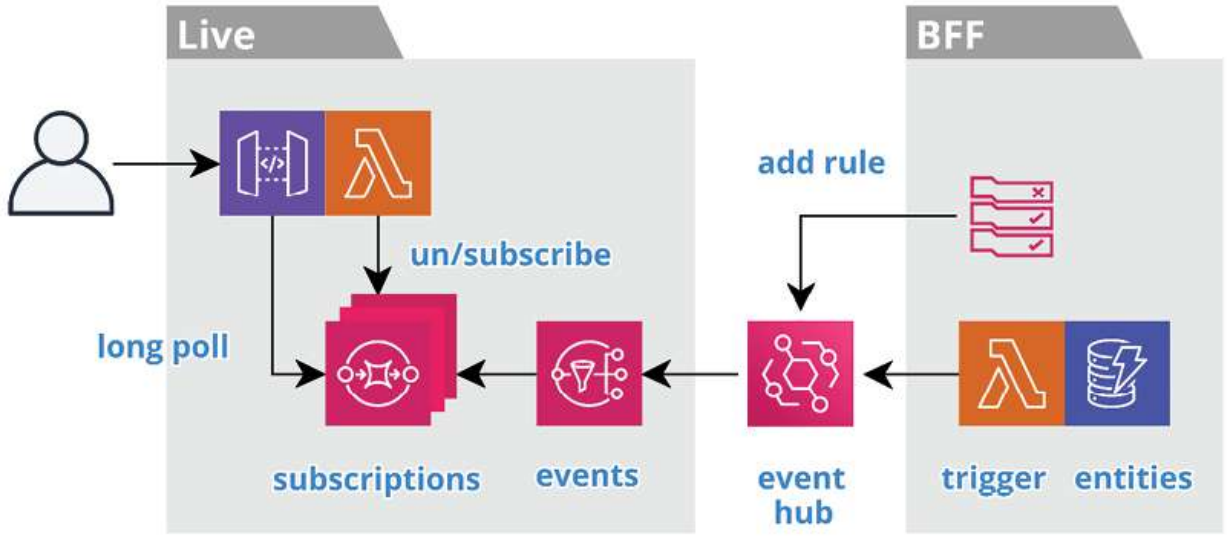


*Figure 3.5: Live updates – long polling The right-hand side of Figure 3.5 is identical to Figure 3.4. The difference starts after the **events** topic receives events. The shared micro application subscribes to the **Live** update service and creates a dedicated SQS queue for the current user and registers the queue with the **events** topic. Again, the micro applications can refine the subscriptions with their own filters.*

The shared micro application performs the long polling for the user and will wait up to the SQS maximum of 20 seconds. Any events that arrive in the queue during that interval will immediately flow through the connection in near real-time. From here, the shared micro application makes the events

available to the other micro applications through window event listeners and the micro applications take over, just like in the WebSocket implementation.

When the user goes offline, the shared application stops polling. In the meantime, events collect for the users in their dedicated queues. When the connection is re-established, the long polling will resume, and the buffered events will flow in and update the cache.

Summary

In this chapter, you learned about many of the recent innovations in the presentation tier, such as SPAs, Jamstack, and deploying at the edge of the cloud. We learned how to decompose the presentation tier using the micro frontends approach to create autonomous micro applications that allow us to tame the presentation tier and easily extend the system by adding new applications instead of modifying existing applications. We also dug into the details and learned advanced state management techniques that allow us to create PWAs that work offline and smoothly support regional failover.

In the next chapter, we will cover **eventually consistent systems** and see how **turning events into facts** promotes autonomy and resilience. We will create an event hub, dissect the Event Sourcing pattern, and learn how to implement stream processors.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below: https://discord.gg/B8TubSxc35

4

Trusting Facts and Eventual Consistency

In *Chapter 3, Taming the Presentation Tier*, we covered the *micro frontend* approach to decompose and decouple monolithic frontend applications. We also covered *Offline-first* and *Mobile-first* thinking and found that, increasingly, mobile users are driving the necessity for *event-first* thinking and eventually consistent systems.

Now, we turn our attention to the asynchronous, inter-service communication patterns that form the foundation for an architecture that enables change. We will see how an *event hub* sits at the heart of each autonomous subsystem and creates an outbound bulkhead that protects upstream autonomous services from downstream outages. Also, we will see how the *event sourcing* pattern turns events into facts and gives teams the confidence to trust eventually consistent systems. These are the foundational elements that allow us to drive down lead times and accelerate the pace of innovation.

In this chapter, we're going to cover the following main topics:

- Living in an eventually consistent world
- Publishing to an event hub
- Dissecting the Event Sourcing pattern
- Processing event streams
- Designing for failure
- Optimizing throughput

Living in an eventually consistent world

In *Chapter 1, Architecting for Innovation,* and *Chapter 2, Defining Boundaries and Letting Go,* we discussed the importance of event-first thinking and decomposing a system into an event-first topology of autonomous services and subsystems. In *Chapter 5, Turning the Cloud into the Database,* we will turn our databases inside out and replicate derived data across the system. We do all this in an effort to create an architecture that enables change. We will fortify the boundaries between services and minimize the need for inter-team coordination so that autonomous teams can experiment, learn, and adapt more quickly.

However, sooner rather than later, in any conversation about loosely coupled, event-driven systems, questions and concerns naturally arise regarding data consistency. These concerns stem from the days of information silos, long batch cycles, and spaghetti integrations that produced reconciliation nightmares. The relational database and real-time integrations were supposed to solve these data consistency problems, but instead, they produced highly coupled systems that are difficult to change and prone to cascading failures.

Data is an information system's most important asset. It must be accurate. However, instead of controlling data too tightly, we want to embrace the fact that we live in an eventually consistent world. Long before there were information systems, the world was able to transact business. We simply need to mimic those traditional processes and make them work in near real time.

The typical coffee shop provides the perfect example. On a slow day, when you are the only customer, the coffee shop owner can take your order, prepare your coffee, receive your payment, and finally, give you your coffee. Everything is nice and consistent. However, on a typical busy morning, this single-threaded, sequential process will completely break down. The line of customers will stretch out the door. Potential customers will avoid the long line, and unhappy customers will not return.

Instead, the typical busy coffee shop is governed by a well-designed and highly efficient *eventually consistent* business process. It is divided into stages, it is fault-tolerant, and it supports concurrency and parallelism. Let's dig deeper into this example.

Staging

The typical coffee shop business process is divided into a set of stages. You place your order with the cashier, and you receive a receipt, which states the *fact* that you paid for your order. Then, you go and stand in the waiting area and monitor the progress of your order.

The cashier hands your order off to the next station and is now free to address the next customer. If you ordered multiple types of items, then your order fans out to multiple stations.

With everything running smoothly, your order works its way through the stations, and one by one, you receive the items in your order. Meanwhile, other customers are moving through the same process, instead of waiting for your order to complete before theirs can start.

This process is an example of **Staged Event-Driven Architecture** (**SEDA**). We have divided the process into a set of stages (that is, stations), and each

stage has an input queue. The queues improve the rate of throughput by minimizing the need for one stage to wait on another. Upstream stages do not need to wait on downstream stages, and downstream stages are never idle unless there is no work.

In a similar fashion, we will utilize SEDA at multiple levels in our software architecture. At the *service level*, the event hub provides the queues (that is, streams), and the autonomous services perform the work at the different stages of a process. We will cover the event hub shortly in the *Publishing to an event hub* section.

At the *function level*, the individual stream processor functions within the autonomous services will implement their own staged process flows. They will leverage **asynchronous non-blocking I/O** to provide the queues that enable multi-tasking and **Functional Reactive Programming** (**FRP**) to define the stages of work. We will cover these topics in the *Processing event streams* and *Optimizing throughput* sections.

Cooperative

The notion of *cooperative programming* is also in play here, at both the function and service levels. The amount of work performed at each stage should be similar. Otherwise, we are creating an unbalanced process flow, where a quick stage can overwhelm the next stage, or a slow stage can starve the next. There is no particular way to enforce this other than through intentional design and inter-team cooperation, hence the term *cooperative*.

Atomic

Each stage must also perform an atomic unit of work. In other words, each stage should update a single resource. This will allow us to maximize

parallelism. It will also minimize waste when something goes wrong, since we will not need to repeat successful stages. Ultimately, we will chain together a series of atomic actions (that is, stages) to achieve eventual consistency.

Consistency

Everyone is happy in our coffee shop example when everything is running smoothly. You have to wait for your order, but you have a receipt that states the *fact* that you have made your payment. You understand that you will *eventually* receive your coffee, and the shop understands that it must fulfill your order or give you a refund.

All the participants agree on how the process works and understand that a little *inconsistency* produces the best outcome for all involved. The process works because we have transparency, facts, and a chain reaction.

Transparency

Transparency is essential in an eventually consistent system. Users need to understand the process, so they can set their expectations. Also, they need to know the state of the system so that they can make informed decisions.

In *Chapter 3, Taming the Presentation Tier*, we saw how, increasingly, mobile users are driving the necessity for eventually consistent systems. Mobile devices lose connectivity, but users can continue working, even though the system becomes inconsistent. Users understand this reality, and the system provides them with visual cues to the current situation.

When a user is consuming data, the system needs to clearly show how old and potentially stale the data may be. If the user is producing data, then it

needs to be clear when the data was successfully submitted. This way, we know when we are working in offline mode, and we understand that everything will eventually become consistent when we reconnect.

Facts

Facts are the lifeblood of an eventually consistent system. Each stage produces an event to record the fact that it completed an atomic unit of work. These facts are not ephemeral. We retain them indefinitely. They represent *data in motion*.

In total, these facts represent the current state of the system. Downstream services may be temporarily inconsistent, but the facts contain all the information needed to make them consistent. The system as a whole is not inconsistent; it just may have more work to complete.

We will see how to turn events into facts in the *Dissecting the Event Sourcing pattern* section.

Chain reaction

We achieve eventual consistency by chaining atomic actions together. Each atomic action produces a fact (that is, an event), and the next atomic action in the chain reacts and produces its own fact. This chain reaction continues until we achieve the desired result. The process is not haphazard; it does not happen by chance. We specifically design the chain reaction to ensure that the system becomes consistent.

We will cover different approaches to control a chain reaction, such as choreography and orchestration, along with compensation, in *Chapter 8*, *Reacting to Events with More Events*.

At any given point, we can review the facts, assess the current state of the system, and determine whether there is any outstanding work to complete. When everything is running smoothly, a chain reaction will happen in near real time. However, things will go wrong, and we need to be prepared.

Concurrency and partitions

In our coffee shop example, everything seems to be running smoothly. You have your breakfast sandwich, but they have lost your coffee. The crew knows how to handle this situation, so they insert a new coffee for you at the head of the queue. You have to wait a little longer than you planned, but you get your coffee. Meanwhile, they identify your original coffee. The name on the label was smudged. Now, you get to choose. You can have the first coffee, the second, or both.

The *CAP theorem* shows us that in the case of a system partition, we must choose between *consistency* and *availability*. We cannot have both. Users clearly favor availability and expect us to ensure that everything eventually comes together cleanly. This is easy enough when no other users are operating on the same logical data. However, we have to account for *concurrency*. We must consider what happens when users interact with the same logical data concurrently.

We have many opportunities for partitions in our systems. Offline mode is an obvious example. Downstream services experience a brief partition as upstream events make their way through chain reactions, and these partitions can be protracted under error conditions. Another example exists when we have two upstream services operating on separate instances of the same logic data.

Our missing coffee is an example of concurrency during a partition. A second person produces a duplicate unit of work, even though it turns out that another person is working on the original. In this case, duplicate output is better than no output. We just need to design the next stage to account for duplicates.

This scenario highlights the importance of *facts* in an eventually consistent system. Multiple users can simultaneously operate on different instances of the same logical data, but we record the result of each action as a fact so that we do not lose information.

One user's change does not necessarily have to overwrite the other. Downstream, we may design the data model to support both changes, weave them together, or drop one or the other. Regardless, we retain the facts as an audit trail.

Inevitably, the facts speak for themselves. We will see how to turn events into facts in the *Dissecting the Event Sourcing pattern* section.

Order tolerance and idempotence

Our coffee shop is order-tolerant. Customers stand in line to place an order, but from that point on, the order in which work leaves the system will vary. Some items are easy to prepare and pass through the system quickly, while others take longer. Mistakes will happen, and items will be re-queued. Also, as we have seen, duplicates can enter the system. A customer waiting for multiple items can receive them in any order. The employees see the individual items as they pass through their stations, and the customers sort out their orders at the other end.

Many messaging systems provide **first-in, first-out** (**FIFO**) guarantees. When things are running smoothly, this can help improve the efficiency of the system. But, as our coffee shop example shows, we cannot rely on these guarantees. Messages may enter the system in the wrong order, others will fail and get re-queued, and multiple channels can work at different paces.

Our stream processors need to act like the customers in our coffee shop and sort things out as they receive events. We do this naturally as customers. We will see how to implement stream processors in general in this chapter. We will cover specific order tolerance and idempotence techniques in the *Implementing idempotence and order tolerance* section of *Chapter 5, Turning the Cloud into the Database*, and in the *Calculating event sourcing snapshots* section of *Chapter 8, Reacting to Events with More Events*.

Parallelism

A line begins to form as customers arrive at the coffee shop. If they arrive faster than we can serve them, then the line will overflow the shop and customers will avoid the line. We have already improved our throughput by dividing the process into stages and allowing different order items to fan out to different stages. However, we can do better, a lot better, by adding another cashier and a second coffee station. This is classic queuing theory at work.

Let's say that customers are arriving at a rate (λ) of 40 per hour, and on average, it takes 1.2 minutes to service a customer, at a rate (μ) of 50 customers per hour with a single queue. Following *Little's Law*, we can calculate that a customer can expect to wait (W) for 6 minutes using the equation $W=1/(\mu-\lambda)$. Also, we can also calculate that, on average, there will be four customers in line (L) using the equation $L=\lambda W$.

However, as the arrival rate increases and we are near the full utilization of our service rate, these numbers will jump significantly. For example, if the rate of customers increases to 48, then the length of the line will jump to 24, and customers will wait for 30 minutes.

In this case, we can open the second queue (that is, the cashier and coffee station) to increase our throughput. We will see both the wait time and the line drop down to approximately 1. A slight modification to our wait time formula bears this out: $W=1/(c\mu-\lambda)$, where c is the number of queues.

Of course, these equations are based on long-term averages, but in practice, the results can be quite astounding. Our eventual consistency systems are based on queues, which makes parallelism a natural fit. We will see the role that *asynchronous non-blocking I/O* and sharding play in the *Optimizing throughput* section.

The reality is that our systems live in an eventually consistent world. Some part of almost every system is eventually consistent. Mobile users need the ability to work in offline mode, which means they are eventually consistent. Integrations with legacy and external systems are eventually consistent and potentially even batch-oriented. Instead of imposing a strict consistency model on an eventually consistent world, it is better to embrace eventual consistency throughout. The impedance mismatch of switching between the different models actually slows us down and impedes innovation.

Let’s see how we put this into action; we’ll start at the beginning with publishing events.

Publishing to an event hub

The ability to publish events is a fundamental capability of autonomous services. Upstream services publish events as their state changes and downstream services react to these events. This publish and subscribe paradigm decouples producers from consumers, but if we are not careful, we can inadvertently introduce coupling and complexity through the messaging infrastructure.

In *Chapter 2, Defining Boundaries and Letting Go*, we discussed the importance of creating an architecture of autonomous subsystems. Each subsystem is composed of a highly cohesive set of autonomous services.

We have seen *Figure 4.1* before. It depicts how we can compose an event-first topology out of autonomous subsystems. This natural fractal pattern produces a system that is very easy to comprehend and reason about. We will see how to connect subsystems in *Chapter 7, Bridging Intersystem Gaps*:

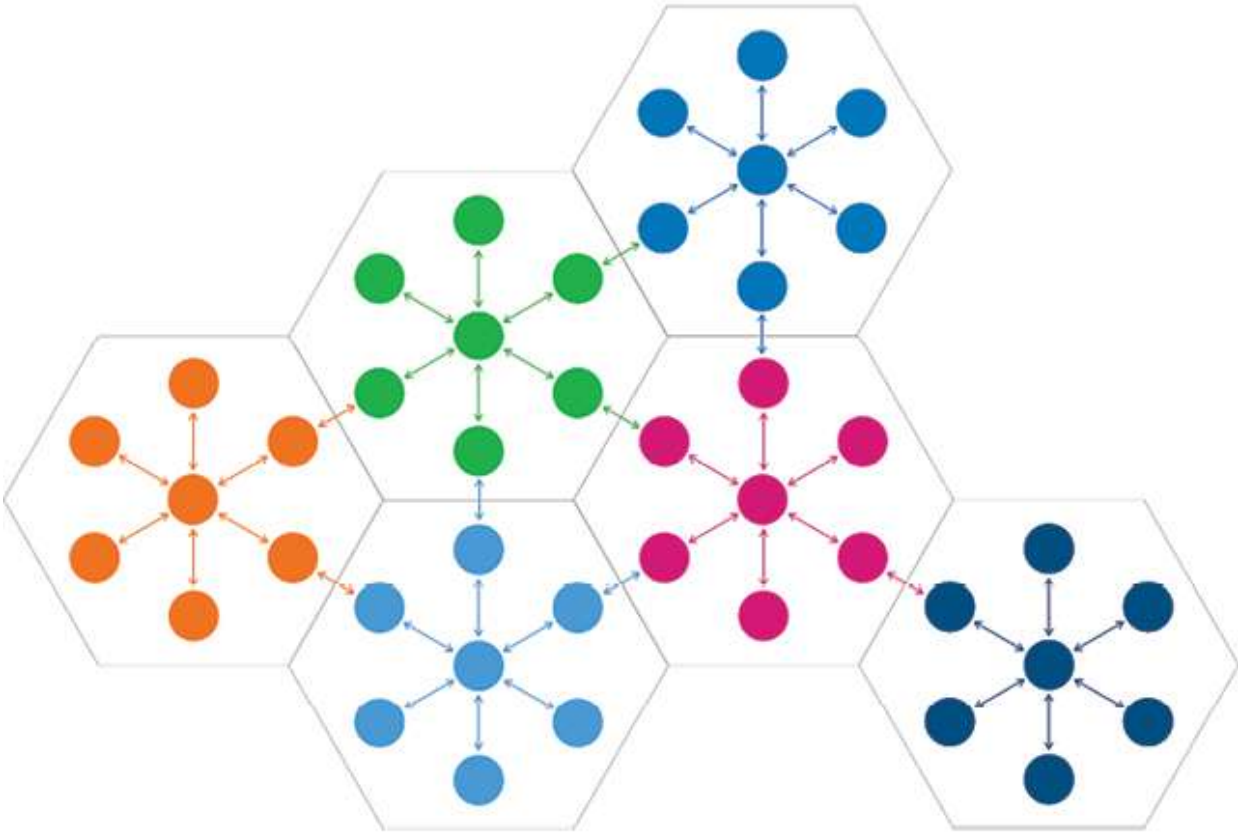


Figure 4.1: Event-first topology

At the heart of each autonomous subsystem is an **event hub**. The event hub eliminates the mayhem that would ensue if we were to create a patchwork of topics and queues to connect the various services. Instead, *Figure 4.2* depicts how the event hub mediates between upstream producers and downstream consumers:

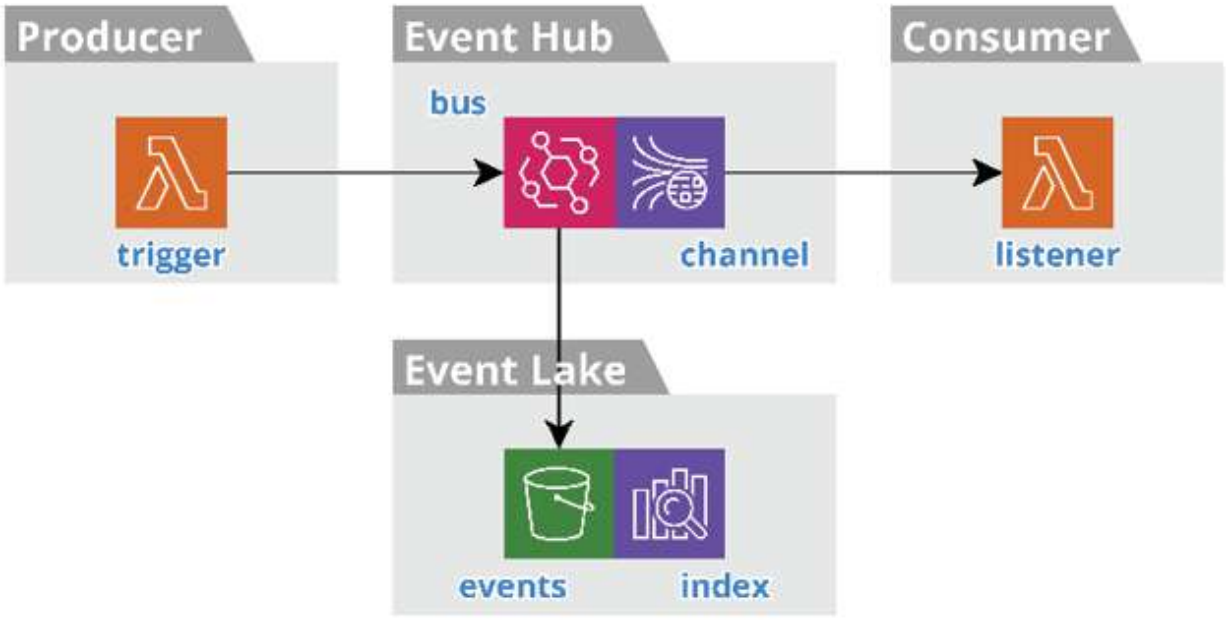


Figure 4.2: Event hub

The general flow of events through the event hub proceeds as follows:

1. Upstream services publish domain events to the hub through a **bus**.
2. The bus routes events to one or more *channels*, such as a stream.
3. Downstream services consume events from the hub through a specific **channel**.

We also see that the **event lake** hooks into the **event hub** to consume all events. This is a crucial part of turning events into facts. We will cover this shortly in the *Dissecting the Event Sourcing pattern* section. First, let's look more closely at the event bus and the format and contents of events.

You can find an event hub template here: https://github.com/jgilbert01/templates/tree/master/template-event-hub.

Event bus

The event bus is the entry point into the event hub. It provides a level of indirection so that we do not couple producers to specific consumer channels. Producers only need to know about the bus. We will add rules to route events to specific consumer channels without impacting the producers.

The event bus also acts as the *outbound bulkhead* that protects upstream services from downstream failures and outages. The bus is a highly available, serverless service, such as AWS EventBridge or Azure Event Grid. An upstream service can publish an event to the bus and forget about it. It can trust that the bus will eventually deliver the event to all interested consumers, through the various channels (that is, streams).

Domain events

Domain events are the lifeblood of an event-first system. We use them to transmit information between services. An actor interacts with a service to perform an action on a domain model. The service produces a domain event to record the fact that the actor performed the action. Then we send the domain event from the producer service to all the consumer services via the event hub.

To support this flow of information, we need a standard event envelope, we need to include the domain model state in each event, and we need to consider the substitution of producers and consumers, both internally and externally.

Event envelope

The event envelope defines a standard set of fields that all events must contain. This allows the bus, any channel, and all consumers to handle any

event. The bus relies on these fields to perform content-based routing to specific channels.

Consumers utilize these fields to filter and dispatch events within their stream processing pipelines. Also, the event lake uses these fields to organize events for storage and retrieval.

Each domain event is a JSON object. The following code block defines the structure of the `Event` envelope:

```
interface Event { id: string; type: string; timestamp: number;
partitionKey?: string; tags: { [key: string]: string | number };
// <entity>: any; raw?: any; eem?: any; }
```

The `Event` envelope defines the following fields:

- The `id` field contains the unique identifier for the event. This value must be deterministic. We cover this in the *Implementing idempotence and order tolerance* section of *Chapter 5*, *Turning the Cloud into the Database*.
- The `type` field indicates the type of the event and the expected payload. It will typically consist of a domain entity name and the action performed, such as `thing-submitted`, along with a namespace prefix for external domain events.
- The `timestamp` field contains the epoch value, in milliseconds, for when the actor performed the action that produced the event.
- The `partitionKey` field allocates events to shards for parallel processing. It typically contains a domain entity identifier or a correlation key, so stream processors can group related events. Otherwise, use a V4 random UUID to ensure even distribution across the shards. We will cover sharding in the *Optimizing throughput* section later.

- The `tags` field contains general information, such as the account, region, stage, source, function name, and pipeline.
- The `<entity>` field contains the event-type-specific payload. The name of this field is usually the *lowerCamelCase* name of the domain entity type, such as `thing` for *Thing*. We will discuss the contents of this field next in the *Event-carried state transfer* section.
- The `raw` field contains the raw data produced by the source of the event, in its native format. We include this data, when available, so the event lake can form a complete audit trail with no lost information. For example, an ESG service may not use all the external data, but it is available here if the requirements change. There are no backward compatibility guarantees for the structure of this data.
- Include the `eem` field when an event contains sensitive data. This field holds the envelope encryption metadata that permitted consumers can use to decrypt the values. We cover this in the *Redacting sensitive data* section of *Chapter 10*, *Securing Autonomous Subsystems in Depth*.

The following JSON object provides an example of a `thing-submitted` domain event:

```
{ id: 'ab3ba422-...-362b9e155667', type: 'thing-submitted',
timestamp: 1574100988321, partitionKey:
'edac237c-...-2b1a3baeccce', tags: { source: that-bff-service-
prd', functionname: 'that-bff-service-prd-trigger', } thing: { id:
'edac237c-...-2b1a3baeccce', name: 'Thing One', description: 'This
is thing one.' } }
```

- The `<entity>` field (that is, `thing`) carries the state to downstream services and represents the contract between producers and consumers. Now, let’s look at this payload in more detail.

Event-carried state transfer

Our domain events are more than just notifications. They represent the fact that something happened, and they contain the data that provides the context of the event. Downstream services use this data to make decisions and create materialized views. We refer to this as the **event carried state transfer** pattern. We already touched on this topic in the *Thinking about events first* section of *Chapter 2*, *Defining Boundaries and Letting Go*.

The `<entity>` field in the standard event envelope represents the payload of a specific event type. It contains a *snapshot* of all the relevant data that is available in the service when the actor performs the action that produces the event.

For example, two BFF services may provide different actors with the ability to work on different aspects of the same domain entity. Both services have a lean, read-only copy of the domain entity's summary information, but each is responsible for maintaining different subsections of the domain model.

When the BFFs publish events, they include the summary information for context, along with the detailed information they own. However, each one does not need to maintain copies of each other's subdomain model and include it in the events. Instead, downstream services as necessary will merge the subdomains in their own materialized views to suit their needs. We will cover materialized views in detail in *Chapter 5, Turning the Cloud into the Database*.

Next, we need to ensure that we do not couple upstream services to downstream services through the contents of an event.

Substitution

We need the ability to change event producers without forcing consumers to change. In other words, a downstream service should be able to consume the same domain event regardless of what service produced it.

Over time, producers may change and, at any given time, there may be multiple sources producing the same domain event. This is the substitution principle that we introduced in *Chapter 2, Defining Boundaries and Letting Go*.

To support substitution, the `<entity>` field of a specific event type should conform to a standard canonical format for the specific domain entity. This format defines the contract between producers and consumers.

The contract does need to evolve, particularly in the early stages of development. To balance flexibility with stability, we will maintain internal and external domain events.

Internal versus external

In *Chapter 2, Defining Boundaries and Letting Go*, we discussed the benefits of creating separate event types for internal and external domain events. Internal domain events define the contracts within a subsystem, and external domain events define the contracts across subsystems.

We give **strong backward compatibility guarantees** to downstream subsystems for the external domain events. This allows teams to experiment and change the definition of the internal domain events more freely. We will see how ESG services create an anti-corruption layer by transforming between the internal and external domain events in *Chapter 7, Bridging Intersystem Gaps*. We will cover the versioning of domain events in the *Achieving zero-downtime* section of *Chapter 11, Choreographing Deployment and Delivery*.

Routing and channel topology

Routing is the purpose of the event hub. Routing helps decouple producers from consumers. Producers send events to the bus which is responsible for routing them to the various consumer channels.

A consumer generally subscribes to a single channel. Many consumers can share a channel. The hub typically owns shared channels and the associated routing rules, whereas a service will own a dedicated channel and add its own routing rules to the bus.

The following `serverless.yml` fragment, from the event hub template, creates an AWS EventBridge routing rule for an Amazon Kinesis stream:

```
Stream1EventRule: Type: AWS::Events::Rule Properties:
EventPattern: detail: type: [ anything-but: fault ] Targets: -
Arn: Fn::GetAtt: [ Stream1, Arn ] InputPath: $.detail ...
```

The `detail` field in the rule represents the event envelope, and we use it to access the content of the event. `EventPattern` includes all event types, except for `fault` events. The rule passes the event envelope (that is, `$.detail`) to the stream via `InputPath`.

A subsystem may need a topology of multiple channels to support the flow of events to downstream consumers. There are a variety of things to consider when defining a topology of channels, such as isolation, message size, priority, volume, and the number of consumers.

It may be necessary to isolate specific channels to ensure that they do not interfere with each other. The event lake, which we will cover in the *Dissecting the Event Sourcing pattern* section, will maintain its own channel and routing rule to ensure that it consumes all events. In *Chapter 7, Bridging Intersystem Gaps*, we discuss the value of *isolating* external events in a

separate ingress channel and using egress routing rules to connect subsystems.

The size of events is important. The routing rule we defined in the preceding paragraphs weeds out fault events. This is an example of excluding a specific event type based on the potential message size. Channels have capacity limits, and large events can impact the throughput of other events. These large events may warrant a dedicated channel. As an example, the fault monitor that we will cover in the *Designing for failure* section maintains its own channel for just fault events.

The priority of events is another consideration. We may define a high-priority channel for one or more event types to avoid competition with low-priority event types. For example, a specific consumer may define a dedicated channel for a specific event type to ensure that there is no competition.

The volume of events is also a consideration. In the *Processing event streams* and the *Optimizing throughput* sections, we will discuss the value of multiplexing many event types through a single channel and filtering on the consumer side. However, some consumers may only require a small fraction of the events. So, it may make sense to have a separate channel for low-volume event types.

The number of consumers may also have an impact on throughput. More consumers on a single channel creates competition for the read capacity, and this can result in read throttling. In this case, if we cannot add more capacity to the channel, then we can create duplicate channels and spread the consumers across the channels.

The routing rule we defined earlier acts as a default channel because it includes all events except for faults. I typically start with this single default

channel and refine the rules and channels, as the subsystem evolves and we learn the exact nature of its domain events and their consumers.

I also prefer to implement channels as streams. They help to turn events into facts and first-class citizens, as we will see in the next section. Also, they provide many performance-related and resilience benefits, as we will see in the *Processing event streams*, *Designing for failure,* and *Optimizing throughput* sections.

Let’s dig into the details.

Dissecting the Event Sourcing pattern

The *Event Sourcing* pattern provides the foundation for eventual consistency and the subsequent flexibility that allows us to create reactive and evolutionary systems. It essentially proposes turning events into facts, instead of just ephemeral messages. We accomplish this by retaining all events and storing them in perpetuity. The events serve as an audit trail of historical activity so that we do not lose information.

Unfortunately, the Event Sourcing pattern has a reputation for making systems more complex. This concern is not unfounded. However, with the right alterations, communicating with events and treating them as facts actually produces systems that are more straightforward and flexible. A few examples will help make this clear. Let’s look at an example without event sourcing, another one with what I refer to as *Traditional Event Sourcing*, and a third example with the altered form, which I refer to as **system-wide event sourcing**.

Figure 4.3 depicts a BFF service that allows users to query their account balance and perform a command to update the balance, without using the event sourcing pattern:

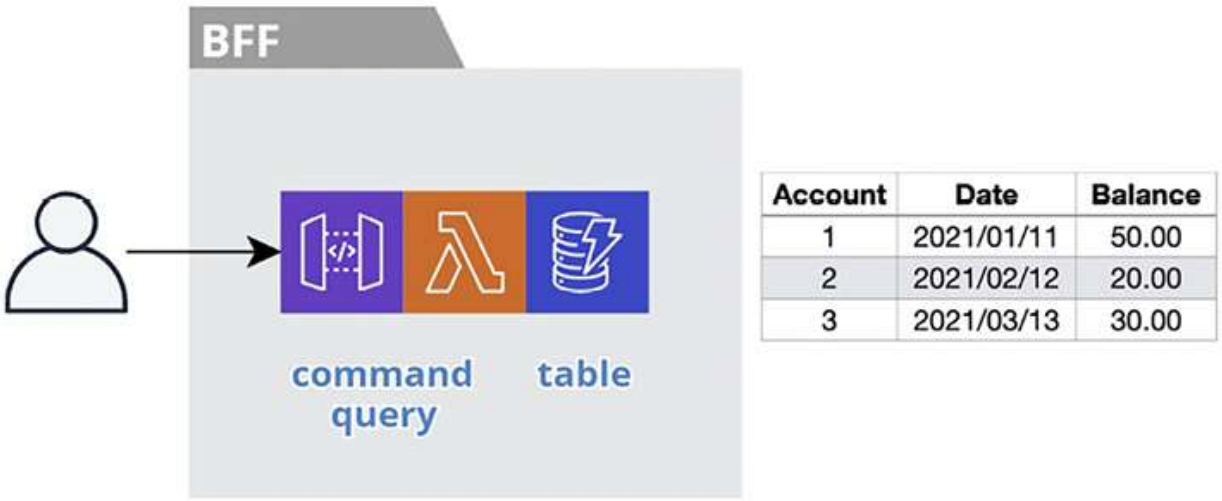


Figure 4.3: Without event sourcing

The functionality is relatively easy and straightforward to implement. We store account balances as a single row per account. The command simply increments or decrements the balance, and the query simply reads the latest state of the balance.

This is a simplistic example, but it highlights an important problem. We are losing information. We only retain the latest balance, so we lose the history of the account. We are not recording the fact that the account holder performed a transaction to increase or decrease the balance.

Figure 4.4 depicts a BFF service that provides the same account balance functionality using the traditional event sourcing pattern:

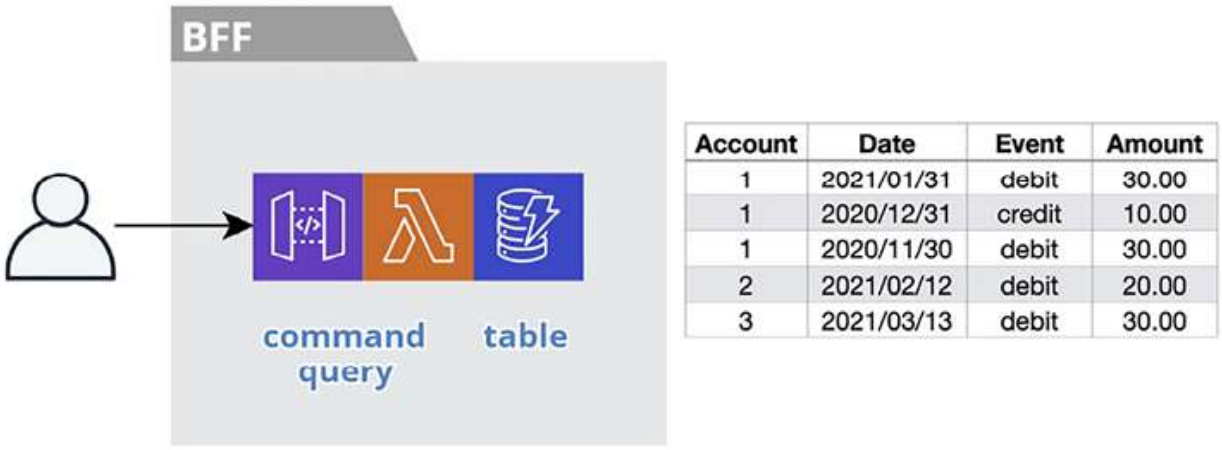


Figure 4.4: Traditional event sourcing

In this case, we do not lose information because we are storing a separate record for each transaction. The command is still straightforward because it just writes a new record, with the amount of the change and a credit or debit flag. However, the query has become much more complex.

The query needs to read all the transactions for an account and calculate the balance on every request. This is not a big problem when there are only a few transactions, but as the number of transactions grows, the performance will suffer. Taking snapshots is a common solution to this performance issue, but snapshotting adds complexity because it adds a component to the design.

Now, let’s look at how we can improve event sourcing.

System-wide event sourcing

The first two examples have one thing in common; they each focus on supporting the usage scenarios of a single actor. This is great from the perspective of the *Single Responsibility Principle*. However, we also need to consider the system as a whole.

Going back to the *Living in an eventually consistent world* section, we saw that an eventually consistent system acts as a series of atomic stages that exchange facts. This simplifies the system as a whole because we design each stage to focus on a single task, and then we string these building blocks together. Furthermore, a stage does not retain facts; it passes them along so that it can focus on performing its job.

With this in mind, we can alter the event sourcing pattern to take on a system-wide perspective. We can keep the simplicity of the first example but enhance it to share facts with downstream services. Also, we can simplify the second example by relieving it of the necessity to retain facts.

Figure 4.5 depicts two BFF services collaborating to support multiple stages in an eventually consistent system. **Service X** provides the command to update the account balance, and **Service Y** provides the query to read the account balance:

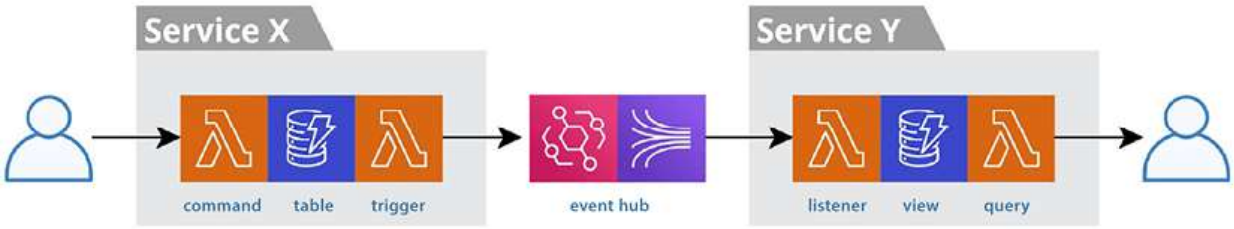


Figure 4.5: System-wide event sourcing

We saw a similar version of this diagram in *Chapter 1*, *Architecting for Innovation*, when we discussed the **Command, Publish, Consume, Query** (**CPCQ**) flow.

The `trigger` function in `Service X` implements the enhancement to our first example. The implementation of its `command` functionality remains straightforward because it only stores the latest balance. We simply add the

`trigger` function to consume the changes from the table's **Change Data Capture** (**CDC**) stream and send these facts to the `event hub`.

The `trigger` function enhancement is a straightforward additive change. We will cover the details of trigger functions in the *Leveraging change data capture* section of *Chapter 5, Turning the Cloud into the Database*. However, so far, we have maintained the simplicity, focus, and atomic nature of **Service X** without losing information because we are sending the facts (that is, events) downstream. We will see how to retain facts in the remainder of this section.

Now, **Service Y** and any other service can sit downstream and react to the facts that upstream services emit. We are gaining flexibility while also retaining information and simplicity. This is the **inversion of responsibility** that we covered in *Chapter 2, Defining Boundaries and Letting Go*. Upstream services just emit events to record the fact that they completed an atomic action, without concern for what happens next. Then, we add downstream services as needed. In other words, event sourcing is a key mechanism to create an *architecture that enables change* because it allows us to change the system by adding services without modifying others.

We will see this example again, in the *Dissecting the CQRS pattern* section of *Chapter 5, Turning the Cloud into the Database*, where we will see how the event sourcing pattern dovetails with the **Command and Query Responsibility Segregation** (**CQRS**) pattern to complete the connection between upstream and downstream services.

Now, let's dig into the different ways that we can retain events and turn them into facts.

Event lake

The event lake is responsible for turning events into facts. It accomplishes this by capturing all events and storing them in perpetuity. It is the ultimate record of truth regarding the actions that have taken place within the system. It is a complete audit trail containing all the facts. At any time, we can dig into the event lake to find the truth about every single state change. In the *Turning the database inside out* section of *Chapter 5, Turning the Cloud into the Database*, we will see how the event lake acts as the *system-wide transaction log*.

We introduced the event lake in *Figure 4.2* in the *Publishing to an event hub* section. At the heart of each autonomous subsystem is an event hub and an event lake. The event lake maintains its own channels and adds its own routing rules to the event hub, ensuring that it receives all events published within the subsystem or consumed from upstream subsystems.

Perpetual storage

The durability of the events within the lake is of the utmost importance. For this reason, *object storage* is the best choice to persist events in the lake. For example, AWS S3 provides 11 nines of durability; plus, we can age events to long-term storage, such as Amazon S3 Glacier. We will also replicate the lake across regions and in a separate account for disaster recovery.

We organize the events in object storage based on their timestamp. We can browse events by year, month, and day, but we package multiple events into a single object for optimal throughput and storage. We can read the individual events with a command-line utility, which we will cover in the *Replaying events* section. However, indexing the events gives us greater visibility into the event lake, as we will see shortly.

The security of events in the event lake is crucial. Maintaining a separate lake per subsystem helps to limit access on a least privileged basis. We also redact sensitive data in the lake using envelope encryption, as we will cover in *Chapter 10, Securing Autonomous Subsystems in Depth*.

A NOTE ON TERMINOLOGY

In my previous books, I referred to the event lake as the data lake. Although an event lake is essentially a data lake, I prefer the new name because it helps highlight the autonomy of the individual event lakes owned by each autonomous subsystem. Each subsystem needs the liberty to replay events from its own event lake at will. Maintaining a single, centralized data lake limits a team's ability to respond quickly. That said, you will most likely create a subsystem dedicated to business analytics. The event lake of that subsystem will contain all the external domain events of all other subsystems and play the role of the corporate data lake.

You can find an event lake storage template here: https://github.com/jgilbert01/templates/tree/master/template-event-lake-s3. This template uses AWS Firehose as the channel to store events in S3.

Now, let's look at indexing the event lake so that we can easily search for events.

Indexing events

The event lake is a valuable source of information. Retaining events as facts in durable storage is crucial, but we also need the ability to easily find events of interest. For example, we can use the events for root cause analysis when we investigate a defect or incident, and once we determine the root cause, we may need to find the right set of events to replay to repair one or more services.

Indexing events allows us to easily search for events on many dimensions. There are many ways to index events. Events are well suited for a time series database because they have a timestamp and various dimensions, such as type and tags. They are also well suited for a search engine because they have rich, coarse-grained content.

Elasticsearch is uniquely suited for indexing this coarse-grained time series information. We can also think of the event lake as a log file and choose to index the events in a **Software as a service** (**SaaS**) observability and log monitoring system. Amazon Athena is another option when we store the lake in S3.

You can find an event lake indexing template here: `https://github.com/jgilbert01/templates/tree/master/template-event-lake-es`. This template uses AWS Firehose as the channel to index events in Elasticsearch.

Replaying events

Retaining facts in the event lake allows us to replay events. We might replay events to repair a broken service, or to seed a new service or a new version of a service. For example, if a bug in a service causes it to miss some events, then we can replay the events after we deploy a fix. For a new service, we might replay the last several weeks or months of events to bring it up to date.

It is important to point out that we do not republish events because this would broadcast the events to all consumers. We send events to a specific listener function. This is the same listener function that consumes from a channel, such as a stream. So the replay utility needs to wrap the events in the format of the specific channel.

We do not need to filter down to the exact set of events when we replay events to a function. As we will see in the *Processing event streams* section, a `listener` function is a stream processor that already filters for the event types of interest. So we can specify a date range and let the function weed out the unwanted events.

When we replay a range of events, the `listener` function will likely have already processed some of the events when they were published. So there will certainly be duplicates, and any missed events will now arrive out of order. This is one more reason why it is important to ensure that our listener functions are idempotent and order-tolerant. We cover these techniques in the *Implementing idempotence and order tolerance* section of *Chapter 5, Turning the Cloud into the Database*.

You can find the `aws-lambda-stream-cli` library here: https://github.com/jgilbert01/aws-lambda-stream-cli. This command-line utility works with the stream processor framework that we will cover in the *Processing event streams* section.

While the event lake is the record of truth and enables us to replay events, we do not intend for autonomous services to access the event lake in real time. Instead, autonomous services interact with event streams and micro-event stores. Let’s look at these next.

Event streams

An event stream is a messaging channel that acts as a temporal event store for downstream consumers. We leverage streams for all inter-service communication channels. They act as the input queue for the various stages in our eventually consistent system.

Autonomous services listen for events on a stream (that is, channel) so that they can react in near real time. From their perspective, a channel or stream is the source of events. In other words, streams provide temporal event sourcing to downstream services.

We will cover stream processing in detail in the remainder of this chapter and discover all the benefits that stream processing provides, such as batching, sharding, and backpressure. But first, we need to cover the benefits that event streams provide to upstream services by turning events into facts, and the benefits they provide to downstream services by providing concurrency control.

Temporal storage

Streams are different from other messaging channels because they also act as an append-only data store. Whereas most messaging channels treat events as ephemeral messages and discard them once acknowledged, a stream retains events for a prescribed period, such as days, weeks, or months.

This creates a very different mental picture that is critical for creating eventually consistent systems. The simple notion of storing events turns them into facts. We start to treat events as first-class citizens. The results of all the atomic actions leading to eventual consistency (that is, the facts)

become part of the data model. We no longer retain just the latest static image. The data becomes dynamic. It is in motion.

Persisting events gives us confidence that data will flow to downstream services and the system will eventually become consistent. We can tune a given channel to hold events long enough to ensure that its consumers have the time they need to react. This frees upstream services from the responsibility of ensuring consistency. Instead, atomic actions and system-wide event sourcing greatly simplify upstream processing logic.

To implement system-wide event sourcing in an upstream service, we can execute either of the following two variations on the pattern: **stream-first** or **database-first**. Let's look at the stream-first variant next.

Stream-first event sourcing

Distributed transactions have fallen out of favor because the two-phase commit cycle does not scale. This is evident by the fact that many cloud-native services do not support distributed transactions. This means that we cannot reliably save data to more than one data store at a time.

Instead, we create eventually consistent systems by stringing together a series of atomic stages. Each stage is *atomic* because it only updates a *single* data store. But this means that we must choose between writing data to a database or emitting a fact to the event hub. We cannot do both as an atomic action.

This is where the stream-first and database-first variants of the event sourcing pattern come into play. We need to turn events into facts. A service can either send an event to the event hub first and trust that downstream services will react and store the data, or a service can store the data first and

trust that its own trigger function will react and publish the event. We will cover the database-first variant in detail in the *Leveraging change data capture* section of *Chapter 5, Turning the Cloud into the Database*.

Here, we will look at the stream-first variant. *Figure 4.6* depicts the stream-first variant of the event sourcing pattern:

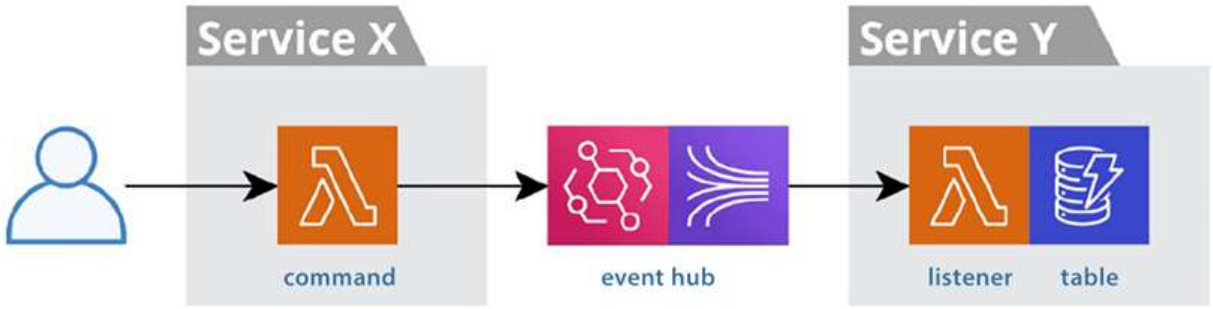


Figure 4.6: Stream-first event sourcing

We have already seen that a stream is a temporal data store. When a `command` sends an event to the `event hub`, we can be confident that the event hub will route it to one or more channels (that is, streams) for storage and consumption downstream. For example, in the *Routing and channel topology* section, we configured a stream as the default channel in the event hub. Downstream `listener` functions consume from these channels and store data for use within their services.

Choosing between the stream-first and database-first variants largely depends on the capabilities of the database in use, plus the user's need to interact with data immediately. If users need to interact with the data they create, then we will lean toward the database-first variant. However, the database-first variant requires a database that supports CDC, such as DynamoDB streams.

In *Chapter 6, A Best Friend for the Frontend*, we will see that BFF services typically employ the database-first variant. However, if the database does

not support CDC, then the commands of the BFF can use the stream-first variant and the BFF can consume its own events and persist the data.

In *Chapter 7*, *Bridging Intersystem Gaps*, and in *Chapter 8, Reacting to Events with More Events*, we will see that ESG and Control services typically employ the stream-first variant because they are not user-facing.

A NOTE ON TERMINOLOGY

In my previous books, I referred to the stream-first variant as event-first. However, since that time, event-first has taken on a new and larger meaning, as we have discussed throughout this book. I also prefer the term stream-first because it is more congruent with the term database-first.

Concurrency control

Concurrency control is another reason that I prefer to insert a stream-based channel between the bus and consumers. For example, Amazon EventBridge has the option to route events directly from the bus to AWS Lambda functions. This option is suitable for very low-volume events, but it can have negative impacts on high-volume events.

First, this direct routing typically delivers a single event at a time to the function. This will increase the number of function invocations with a corresponding increase in cost. This also eliminates the efficiencies we can gain through batching of events, as we will see in the *Processing event streams* section. These inefficiencies will increase the cost as well.

Furthermore, this direct routing delivers events to functions as quickly as possible without regard for the concurrency limits on functions. This will

result in throttling and cause the bus to retry delivery until it succeeds. We will cover concurrency limits in *Chapter 12, Optimizing Observability*.

We can avoid all these issues by simply inserting a stream-based channel. We set the stream capacity so that it can receive events from the bus without throttling. Then, the stream processor function can consume from the stream in an orderly manner. We will cover more benefits in the *Optimizing throughput* section.

Now, let's see how individual services can maintain their own micro-event store so that they can access older events.

Micro-event stores

A micro-event store contains a subset of facts that are important to a specific service. Whereas the event lake is responsible for storing all events and an event stream stores the most recent events, a service can define its own micro-event store to provide dedicated access to a subset of events.

The service collects events of interest and correlates them in a micro-event store, such as a DynamoDB table. This gives the service optimized access to retrieve these related events. It retains these facts only as long as necessary by setting a time to live on each record.

In *Chapter 5, Turning the Cloud into the Database*, we will see the role a micro-event store can play in idempotence and order tolerance. In *Chapter 8, Reacting to Events with More Events*, we will see how we can use a micro-event store to correlate events for complex event processing and business process orchestration. Plus, we will use a micro-event store to calculate event sourcing snapshots.

Now, let’s cover stream processors and learn how to implement listener and trigger functions.

Processing event streams

At first glance, stream processing might seem like it is some extreme paradigm shift. And truth be told, if you have to do it all from scratch, it is not for the fainthearted. For example, we need a process that consumes from a stream and invokes the business logic. We need one of these for each shard. We need logic to increase or decrease the number of these processes with the number of shards. We need to restart these processes when they fall over. Finally, we need to keep track of the offset so that we read from the right position on the next call.

This can add up to a decent cloud bill, not to mention all the development elbow grease required. Of course, we can implement this processing as a container sidecar. But serverless, function-as-a-service offerings, such as AWS Lambda, provide this capability for just the cost of invoking the functions to execute the business logic. It takes care of all the heavy lifting.

Still, stream processing is different, but in a good way. The difference starts with the batch size, which leads us to functional reactive programming. This, in turn, opens a whole realm of possibilities, such as filtering, mapping, reducing, backpressure, and more.

You can find example stream processors here: https://github.com/jgilbert01/templates/tree/master/template-bff-service.

Let’s dive into creating a stream processor.

Micro batching

The input signature of a function is one of the first things we must consider when we implement a service using **function-as-a-service**. The structure of the input events is dependent on the type of messaging channel feeding the function. For example, an AWS Lambda function consuming from an Amazon Kinesis stream receives a batch of records like the following:

```
{ "Records": [{ "kinesis": { "partitionKey": "1",
"sequenceNumber": "49590338...88898", "data": "SGVsbG8s...N0Lg==",
},... },{ "kinesis": { "partitionKey": "1", "sequenceNumber":
"49590338...5618", "data": "VghpcyBp...zdC4=", },... }] }
```

The `data` field of each record is `base64`-encoded and contains the domain event in the format that we covered in the *Publishing to an event hub* section.

The function will only receive one record if there is only a single record on the stream, or up to the batch size limit specified for the function when there are many records in the stream. We can think of this as micro-batch processing, where each invocation processes the next micro-batch of events on the *service-level* stream.

The following `serverless.yml` fragment, from the example service template, creates a `listener` function and configures it to consume from the Amazon Kinesis stream that we defined as the default channel in the event hub:

```
functions: ... listener: handler: src/listener/index.handle
events: - stream: type: kinesis arn: ${cf:event-
hub-${opt:stage}.stream1Arn} batchSize: 100
```

We have control over `batchSize`. In this case, we set it to `100`. This is significant because it opens the opportunity for the many optimizations we will cover in this chapter, such as backpressure and asynchronous non-

blocking I/O. This is the opposite of many traditional messaging channels that only allow us to process a single message at a time, even though they are batching and buffering at the lower level.

Batch size can also have a significant positive impact on the cost of serverless computing at high volumes because we pay for each innovation of a function. Instead of paying for one invocation per record, the batch size acts as a factor to reduce the number of innovations. And when we apply the many optimizations, we also improve the efficiency of each innovation.

At the *service level*, we have chosen to use streaming technology for our messaging channels. At the *function level*, the batch size allows us to slice up data flowing through the channels and create *function-level* streams within our stream processors. This, in turn, means we need to choose a programming model that is best suited for processing streams.

Choosing a programming paradigm

We need to decide which kind of programming paradigm we will use to process our batch (that is, array) of records. **Imperative programming** is the traditional choice, but it has some inefficiencies in this context. **Functional reactive programming** (**FRP**) is a step in the right direction, but we can do better. Ultimately, we will choose to use **stream processing**, but let’s look at how we arrive at that decision.

Imperative programming

We all know the imperative programming paradigm. It is what we all learned when we first started programming. So it is understandable that we might try

to use this paradigm to implement our stream processing functions. However, let's look at an example to see where this falls short:

```
export const handler = async (event) => { for (const record of event.Records) { if (filterOnEventType(record)) { const updateRequest = mapToUpdateRequest(record); await update(updateRequest); } } return 'Success'; };
```

In this example, we use a `for` loop to iterate through the records in the incoming array. Then, we use an `if` statement to identify the records we are interested in. Next, we map the data to the desired format, and then we update the target.

These are the essential steps of virtually every stream processor. We need to transform the incoming data into another format and then send it to another service, such as a data store or the event hub. This means that our functions are I/O-bound. In other words, we will spend most of the processing time waiting for the I/O calls to complete.

In this imperative programming example, we are looping through the records one by one and waiting for each update call to complete before moving on to the next record. This is very inefficient. It is reasonable to expect these functions to operate at under 10% utilization. We can certainly improve on this, but things will get messy quickly, so let's try another programming paradigm.

Functional reactive programming

Processing an array of records is a natural fit for FRP. The most obvious difference between FRP and imperative programming is the absence of loops. FRP is much more declarative. Here is an example:

```
export const handler = async (event) => { await Promise.all( event.Records .filter(onEventType) .map(toUpdateRequest)
```

```
.map(updateRequest => await update(updateRequest) ); return
'Success'; };
```

In this example, we use the methods provided by the JavaScript `array` class, such as `filter and map`, to iterate over and manipulate the records. This code looks much cleaner. We are no longer waiting for calls one by one, but now, we are doing too much all at once.

The `filter and map` methods return a new array. So we apply the filter to all the records, then we map all the records, and finally, we make all the update calls and wait for `Promise.all` to complete. We have gone from one extreme to the other. This might be OK, but there are some potential problems.

First, we might overwhelm that target with too many calls all at once. We will cover backpressure and rate limiting shortly in the *Designing for failure* section. Plus, we are still spending a lot of time doing nothing but waiting on I/O. So let's see how we can build on FRP and take more control over what happens when we wait on I/O.

Stream processing

Streaming is an overloaded term. We use it at the *service level* when we talk about stream-based channels, such as AWS Kinesis. We also use this term at the *function level* when we need to process a stream of data, such as reading or writing a large file. These two levels come together in our stream processor functions one *micro-batch* at a time.

Node.js has a built-in `Stream` module, but we will use a streaming library on top of this called `Highland.js` (https://caolan.github.io/highland). `Highland.js` preserves the FRP style with features such as `filter`, `map`, and `reduce`, and adds

advanced features such as backpressure, batching, parallel processing, and more. Here is an example:

```
import _ from 'highland'; export const handler = async (event) =>
{ await _(event.Records) .filter(onEventType)
.map(toUpdateRequest) .map((updateRequest) =>
_(update(updateRequest)) .parallel(4) .toPromise(); // consume
return 'Success'; }
```

In this example, a key thing to note is that the code we see here only `assembles` the pipeline of steps (that is, stream objects) that the data will flow through. A good analogy is the plumbing in a house. Plumbers assemble the pipes when the house is constructed, but the water does not flow through the pipes until we open a faucet.

The `Highland.js` stream constructor function initializes the pipeline by wrapping the array of records in a stream object, `_(event.Records)`. Then, we declare the remaining steps that will process data as it flows through the pipeline. This includes the new `parallel` step that allows us to easily control the amount of I/O we perform at once. The final step, `toPromise`, then, and only then, starts consuming data from the *function-level* stream, and the data starts flowing through the processing steps. In other words, we start pulling data through the steps after we have assembled the pipeline.

This is an important distinction. Each step pulls work from the previous step, instead of looping over the data and pushing it downstream. This gives us full control over how and when data flows. For example, while waiting on I/O we can continue to pull data through upstream steps to optimize utilization. The promise will resolve once all the data has passed through all the stream steps or be rejected when we encounter an unhandled error.

This is an example of function-level **Staged Event-Driven Architecture** (**SEDA**), as we covered in the *Living in an eventually consistent world*

section. Each step represents a stage in the processing pipeline. Each stage has an input and output buffer provided by the low-level streaming library. Upstream stages can continue to pull work when downstream stages block. This will be very important when we talk about asynchronous non-blocking I/O in the *Optimizing throughput* section.

Now, let's put our chosen programming paradigm to work.

Creating a stream

Before long, you will notice that our stream processors follow very similar patterns from service to service. I have accumulated a variety of reusable components into a library that provides a lightweight framework to create stream processors. We will use this library in our remaining examples. You can find the open source `aws-lambda-stream` library here: https://github.com/jgilbert01/aws-lambda-stream.

The first thing we need to do in our stream processors is create a stream from the incoming records. The various streaming and messaging channels each have their own formats. We want to decouple the stream processor logic from the choice of these technologies. The first step of a stream processor transforms the incoming records, using the appropriate `from` function, such as `fromKinesis`, `fromDynamodb`, and `fromEventBridge`, to normalize the records into our standard event envelope format.

The following code block shows the `listener` function that consumes from the Kinesis stream:

```
import { fromKinesis, ... } from 'aws-lambda-stream'; export const
handler = async (event) => fromKinesis(event) .filter(onEventType)
.map(toUpdateRequest) .through(update({ parallel: 4 }))
.through(toPromise);
```

This example consumes events from a Kinesis stream and materializes the data in a DynamoDB table. The `fromKinesis` function understands the structure of the Kinesis records and turns the `event.Records` array into a *function-level* stream of events. The remaining steps are unaware that the events came from Kinesis.

This is an example of Hexagonal Architecture, as we covered in *Chapter 2, Defining Boundaries and Letting Go*. The `from` functions are adapters that hide our design decisions regarding which channel technology we use. We can easily change our minds and substitute a different `from` function, such as `fromSqs`. We won't need to change any of the business logic downstream, because we code it to the event envelope that we covered in the *Publishing to an event hub* section instead of the channel-specific structures.

Now, let's cover immutability and unit-of-work objects.

Unit of work

More so than ever, we do not use global variables in stream processors. Instead, we employ immutable data structures in FRP and stream processing. The data we pass between the steps of the pipeline must be immutable. This ensures that the various stages do not interfere with each other. Without immutability, it would be virtually impossible to reason about the current state of the data because we do not have explicit control over when each stage pulls the next unit of work as the events flow through the pipeline.

To help us achieve immutability, we pass **unit-of-work** (**uow**) objects through the pipeline. Think of a `uow` object as an immutable object that represents the lexical scope of a set of variables passing through the stream. Your processor steps will adorn their outputs to the `uow` for use by downstream steps. We will see how to do this in the *Mapping* section.

The various `from` functions return a stream of `unit of work` objects. The following code block defines the base structure of a `UnitOfWork`:

```
interface UnitOfWork { record: any; event?: Event; batch?:
UnitOfWork[]; }
```

A unit of work starts with the following fields:

- The `record` field contains the original record from the messaging channel.
- The `event` field contains the standard event envelope that we extracted from the original record.
- The optional `batch` field contains an array of related units of work that should succeed or fail together. We will see how to use this field when we discuss batching and grouping in the *Optimizing throughput* section.

The scoping provided by a unit of work will be crucial when we leverage the various parallel processing and pipeline features of the stream processing framework.

Now, let's filter for events of interest.

Filtering and multiplexing

In the *Routing and channel topology* section, we addressed the many considerations that go into defining the messaging channels within an autonomous subsystem.

Achieving an optimal topology can be a balancing act. Inevitably, we will need to multiplex different event types through the same channel, as we will see in the *Optimizing throughput* section. Also, it is reasonable to start with a single channel for all event types and then evolve.

Filtering decouples a stream processor from these decisions. It weeds out unwanted event types. We specify the event types of interest, and the processor ignores any others. As the messaging channel topology evolves, we simply reassign the function to a different channel. We do not need to update the filters or any other logic in the stream processor.

The following code block defines the `onEventType` filter we use in our example `listener` function. It uses a regular expression to match all event types for the `Thing` domain entity. In other words, this `listener` function will process all events where the value of the `type` field starts with `thing-`:

```
const onEventType = uow => uow.event.type.match(/thing-*/);
```

A `filter` function can be arbitrarily complex. It is helpful to create layers of filters. The first filter weeds out unwanted event types, and then the next filter dispatches events to different pipelines based on the content of the event. An individual pipeline can have its own filters. We will cover pipelines in the *Optimizing throughput* section. Ultimately, a filter guards downstream steps against irrelevant events so that they can focus on their responsibilities.

Now, let's transform the events of interest into the desired format.

Mapping

Transforming data from one format to another is the primary job of a stream processor. It is not glamorous work, but it is nevertheless important. A `listener` function consumes upstream domain events and creates materialized views so that end user queries do not need to perform transformations over and over again. A `trigger` function transforms local state changes into domain events for downstream consumption.

The following code block shows the basic form of a mapping step. `uow` must be immutable, so we return a new `uow` object by cloning the original `uow` object with the spread operator (`...`) and adorning the additional variables:

```
.map((uow) => ({ ...uow, variableName: { // mapping logic here }
}))
```

The following code block defines the `toUpdateRequest` mapping function we use in our example `listener` function. It is responsible for preparing the data for the *connector* step. It transforms the domain event into the required format and assigns it to the `updateRequest` variable that the downstream connector step expects:

```
import { updateExpression } from 'aws-lambda-stream'; const
toUpdateRequest = (uow) => ({ ...uow, updateRequest: { Key: { pk:
uow.event.thing.id, sk: 'thing' }, ...updateExpression({
...uow.event.thing, timestamp: uow.event.timestamp, }), } });
```

A mapping function can be arbitrarily complex. In this case, we prepare to update a materialized view in DynamoDB. There is plenty of opportunity to reuse utility functions, such as the `updateExpression` function, which transforms a plain old object into a DynamoDB update expression. We will cover the details of creating materialized views in *Chapter 5*, *Turning the Cloud into the Database*.

It is important to perform mapping in a separate upstream step from the connector step that will perform the asynchronous non-blocking I/O work. This is an example of *cooperative programming*, which helps maximize the potential for concurrent processing while our connectors await a response from external resources.

Now, let’s make those external requests.

Connectors

At the tail end of a stream processor, there is a sink step that records the output of processing a unit of work. This sink step might persist data in a materialized view, call an external service, or publish a new domain event. The possibilities are endless, but the pattern remains the same.

We wrap these calls in thin connector classes so that we can easily create mocks for testing. We will dig deeper into model and connector classes in *Chapter 6, A Best Friend for the Frontend*. We wrap a connector with a reusable function to integrate it into the streaming framework.

The following code block shows the reusable `update` function we are using in our example `listener` function. It wraps a connector for DynamoDB and calls the `updateItem` operation to write the data to a table. We will cover the details of creating materialized views in *Chapter 5, Turning the Cloud into the Database*:

```
import { update, ... } from 'aws-lambda-stream'; export const
handler = async (event) => ... .through(update({ parallel: 4 }))
.through(toPromise);
```

These reusable functions leverage *currying* to override default configuration settings, such as `batchSize` and the number of `parallel` executions. In this example, we are overriding the `parallel` configuration. We will cover the `batchSize` and `parallel` configurations in the *Optimizing throughput* section.

Here is another example of a `trigger` function. This stream processor consumes change events from a DynamoDB stream and uses the reusable `publish` function to send domain events to the event hub:

```
import { publish, ... } from 'aws-lambda-stream'; export const
handler = async (event) => fromDynamodb(event) .map(toEvent)
```

```
.through(publish({ batchSize: 10 })) .through(toPromise);
```

It is important to note that the stream processors are performing *atomic* actions, as we discussed in the *Living in an eventually consistent world* section. Each is updating a single resource. Each unit of work constitutes a separate *idempotent* transaction against a single resource.

Before we move on to the topic of optimizing throughput, we first need to cover the all-important topic of designing for failure.

Designing for failure

Sooner or later, our stream processors will run into trouble. A target system may go down, poison events may start arriving, or someone may make an honest human error. It will happen, and we must be prepared. We will cover monitoring and alerting in *Chapter 12*, *Optimizing Observability*. In the meantime, we need to prepare for the worst and design our stream processors to be resilient and self-healing.

We will cover idempotence and order tolerance in *Chapter 5*, *Turning the Cloud into the Database*. In the next section, we will look at backpressure, rate limiting, and poison events. And when we do get into trouble, we will emit fault events.

Backpressure and rate limiting

Backpressure is an important characteristic of stream processing. A stream processor should not overload its target resource. In the best case, the target will throw throttling errors, and we will have low throughput, as the stream processor wastes time and resources. In the worst case, we could end up with no throughput if the processor function times out and creates an infinite retry

loop. In the worst case, the stream processor could overwhelm the target system and cause an outage.

As we saw in the *Choosing a programming paradigm* section, when we use plain functional reactive programming, we can easily overwhelm the target resource. The function will process all the records at once without regard for the throughput capacity of the target system. If we send too many requests, too fast, the target could overload.

Stream processing, on the other hand, provides natural backpressure, because it is pull-oriented. In other words, a step only pulls the next unit of work after it has completed the current. This means a slow target resource will naturally push back, based solely on its own latency. The steps in the stream processor will proceed only as fast as the slowest step.

However, cloud-native resources, such as AWS DynamoDB, can process requests with an extremely high throughput. These resources rely on throttling to restrict capacity. In this case, the natural backpressure is not enough. We will need to insert artificial backpressure.

The following code block introduces rate limiting to add backpressure:

```
export const handler = async (event) => ... .rateLimit(2, 100) //
2 per 100ms .flatMap(makeSomeAsyncCall) ...
```

The `Highland.js` `rateLimit` feature acts as a governor to slow the flow of work through the stream processor. In this example, we call the target no more than twice every `100` milliseconds. You will need to know the capacity limits of the specific resource to calculate the proper rate. You will also need to account for multiple shards and parallel calls, which we will cover in the *Optimizing throughput* section.

Poison events

Some errors are transient and self-healing. For example, network hiccups just need a simple retry. We will cover *timeouts and retries* in *Chapter 12, Optimizing Observability*. On the other hand, a transient error could be something more serious, such as an outage in the target system of a stream processor. In this case, the stream processor can let events queue and retry an entire batch until the target system recovers. We can combine this with the *circuit-breaker* pattern that we will cover in *Chapter 7, Bridging Intersystem Gaps*. Also, we will monitor the *iterator age*, which we will cover in *Chapter 12, Optimizing Observability*.

Poison events, on the other hand, are not recoverable. A poison event contains data that breaks the consuming stream processor. It may be a bug in the consuming or the producing stream processor. Regardless, the consuming stream processor will continue to retry the entire batch unless we take corrective action.

Let's say we have a batch of 100 events, and it has a poison event in the 60th position. If the processor does not properly handle the poison event, then it will retry the batch over and over again until the poison event expires from the stream. In the meantime, the first 59 events will update the target over and over again. This is one reason why we need to implement idempotent transactions. We cover idempotence in *Chapter 5, Turning the Cloud into the Database*. And our iterator age monitor will alert us to the problem so that we can react well before the events expire.

The cloud services we leverage may provide support for these scenarios. For example, AWS Lambda provides a *bisection* feature that provides some relief. This will cut the batch size in half before retrying. So, in this example, the first 50 events will retry and succeed. Then, the next 25 will retry, and

the first 9 will succeed again, while the tenth will fail. Then, it will retry a batch of 12 with the same result. And then a batch of six will succeed, but there are still three more good events to go before we narrow in on the poison event. So, the bisection continues until it reaches a batch size of 1 that contains the poison event.

From here, we can send the poison event to a **Dead Letter Queue** (**DLQ**) for special processing. However, when there is one poison event, there are usually many, so this process continues. Meanwhile, the additional latency causes more and more good events to queue in the stream.

We can do better. We can make our stream processor logic more proactive by handling poison events and emitting *fault* events.

Fault events

Stream processors emit fault events when they encounter expected error conditions. A fault is an event type that contains an error and the corresponding event or events that caused the error. We will see in a moment how a stream processor emits fault events by throwing handled exceptions. Then, we leverage the event hub and all its routing features as the *DLQ*, instead of creating and maintaining something special. The fault events flow into the event lake like any other. Downstream, a *fault monitor* service collects faults, meaning that we can resubmit the affected events after we address the root cause.

The following code block defines the structure of a fault event:

```
export const FAULT_EVENT_TYPE: string = 'fault'; interface
FaultEvent extends Event { err: { name: string; message: string;
stack: string; }; uow: UnitOfWork; }
```

A fault event has all the standard fields of the event envelope; the `type` field is set to `fault`, and it includes the following additional fields:

- The `err` field contains the error information, including the name, message, and stack trace.
- The `uow` field contains the unit of work with the state of the variables that were in scope when the error happened, plus the one or more events that were affected.

When a stream processor encounters an error, the error will skip over all the remaining steps until it is either handled or reaches the end of the stream. If it reaches the end of the stream, then all processing stops, and the function returns the error. At this point, the batch will retry, as we discussed in the *Poison events* section.

Instead, we want to write proactive code that handles poison events explicitly. We will set them aside as fault events so that good events can continue to flow. Then, we can delegate the processing of fault events to other components.

The following code block provides an example of a mapping step with error-handling logic:

```
.map((uow) => { try { return { // mapping logic here }; } catch
(err) { err.uow = uow; throw err; } })
```

We wrap the logic in a `try…catch` block. The `catch` block adorns the troubled unit of work to the error and rethrows it. The presence of the `uow` field on the error acts as an indicator that we handled the error and want to emit a `fault` event. The following example uses a utility function to accomplish the same error handling:

```
import { faulty } from 'aws-lambda-stream'; .map(faulty((uow) =>
({ // mapping logic her })))
```

It is a good practice to add a specific validation step at the beginning of a processor to assert that events contain all the required information. This step throws handled errors when data is invalid.

At the tail end of the stream processor, we add the following steps to emit fault events:

```
import { faults, flushFaults, ... } from 'aws-lambda-stream'; ...
.errors(faults) .through(flushFaults) .through(toPromise);
```

The `errors` method is a Highland.js feature to catch errors in a stream. The `faults` function checks `err` to see whether we handled it and adorned a `uow` object. If so, then it buffers a new fault event. We buffer the faults until the end in case an unhandled error occurs later in the batch. Finally, the `flushFaults` step emits all the buffered fault events.

Now, good events can continue to flow, and we avoid a traffic jam. We also delegate fault processing to the fault monitor service and the resubmission process.

Resubmission

Our stream processors now proactively handle poison events and produce fault events so that good events keep flowing. In *Chapter 12*, *Optimizing Observability*, we discuss alerting on fault events so that we can jump into action and resolve a problem quickly. In the meantime, we need to collect fault events so that we can resubmit the affected events once we resolve the problem.

A fault monitor service consumes all fault events from the event bus and stores them in object storage. This is very similar to the event lake, but it makes it easier to access the faults for resubmission. You can find a fault

monitor template here: https://github.com/jgilbert01/templates/tree/master/template-event-fault-monitor. This template uses AWS Firehose as the channel to store fault events in S3.

Once we address the root cause of a fault, we can resubmit the original event back to the stream processor that raised the fault, ensuring that we successfully process all the events. The fault event contains the original event in the unit of work along with the name of the function. So, a utility can read the fault event from the bucket and invoke the specific function with the original event.

You can find the `aws-lambda-stream-cli` library here: https://github.com/jgilbert01/aws-lambda-stream-cli. This command-line utility works with the stream processor framework that we are using in this chapter.

It is important to recognize that we are now processing these events out of order, since we need to set them aside. This is one more reason why it is necessary to ensure that our stream processing logic is order-tolerant. We will cover order tolerance in *Chapter 5*, *Turning the Cloud into the Database*.

Now, let's look at how we can optimize our stream processor logic for optimal throughput.

Optimizing throughput

Throughput is of the utmost importance for a stream processor. Is a specific stream processor keeping pace with the rate of incoming events or is it falling behind? Is the processor overworked or is it spending too much time

waiting? Are we putting too much load on the target resource? There are many ways we can optimize throughput to ensure that we are neither overloading nor wasting resources.

In the *Living in an eventually consistent world* section, we introduced **SEDA** and **Little's Law** and discussed their overall impact on eventually consistent systems. Now, it is time to look at these again from the perspective of individual stream processors.

Let's see how we can divide individual stream processors into stages and implement parallelism to optimize throughput.

Batch size function parameter

Batch size is one of the main parameters we have to adjust the performance of a stream processor. It controls how much work we feed to a processor on each invocation. Should we give it more work and let it run longer, or give it less work and invoke it more often? How does this impact the cost of the stream processor?

As you define and evolve your channel topology, you will need to know the expected volume of traffic for each channel. For each stream processor, you will need to understand the characteristics of the target resource.

From here, you will need to experiment with different batch sizes. As you improve the efficiency of a processor, you may be able to increase the batch size. If you are multiplexing many event types through a single channel, then a higher batch size helps to ensure that your processor has work to perform on each invocation. As you change the number of shards, you may also want to change the batch size. As you increase the batch size, you may also need to increase the function timeout.

Reaching the optimal channel topology and stream processor batch sizes is an iterative balancing act. Keep in mind that the optimal solution also accounts for cost. For example, we could have a channel for each processor, which may optimize the processors at the expense of channel cost and maintenance complexity.

It's a balancing act. I like to start with a single channel per subsystem and a default batch size of 100. Then, I analyze the iterator age metrics and the invocation metrics as I dial in the best configuration. We will cover iterator age in *Chapter 12, Optimizing Observability*.

Now, let's make sure we are not wasting time waiting on blocked calls when we could be making progress on other elements in a batch.

Asynchronous non-blocking I/O

We introduced **SEDA** in the *Living in an eventually consistent world* section. At the service level, a stream processor represents a stage in an eventually consistent process flow. At the function level, a stream processor implements its own staged process flow.

In the *Stream processing* section, we define a series of steps (that is, stages) that the data flows through. The processor pulls data through these steps instead of pushing it. Down inside the streaming mechanisms, each step maintains a buffer (that is, an in-memory queue).

When a step completes work, it places the output in the buffer, and then it pulls the next unit of work from the preceding step.

To optimize throughput, we need to keep these steps busy. We do not want to starve downstream steps, and we do not want to make the output of upstream

steps wait. This is where *asynchronous non-blocking I/O* plays a very important role.

As we saw in the *Choosing a programming paradigm* section, the steps in a stream processor that make an external call (that is, I/O) are typically the most time-consuming. We do not want to waste time blocking on these calls. We want to keep the other steps busy while we wait for a response. We want upstream steps to continue working on the batch and making preparations for more external calls, and we want downstream steps to continue their work.

The following code allows up to eight `parallel` calls to the preceding asynchronous step:

```
export const handler = async (event) => ...
.map(makeSomeAsyncCall) .parallel(8) ...
```

While it waits for a response, the Highland.js `parallel` feature continues to pull work through the upstream steps until it has reached the specified number of calls. For example, an upstream mapping step can prepare the input for the next request, so that we can immediately make another call when we receive a response to a previous call.

Asynchronous non-blocking I/O is probably the most important feature for optimizing throughput. This is usually the first parameter I tweak when tuning a stream processor. The command-line utility we covered in the *Replaying events* section serves as an excellent example.

I was using an early iteration of the utility to replay a year's worth of events from an event lake to seed a new service. It was taking several hours to process just a month's worth of events. It turned out that I was not leveraging the `parallel` feature on all the calls to S3. So, I bumped the

setting up to 16 parallel calls, and performance jumped to just 5 minutes. In retrospect, following *Little's Law*, this makes perfect sense.

Now, let's leverage pipelines to further ensure that we are keeping our processors busy.

Pipelines and multiplexing

In the *Routing and channel topology* section, we covered the many considerations that go into defining the messaging channels within an autonomous subsystem. To achieve an optimal topology, we inevitably need to *multiplex* different event types through the same channel. This actually works to our advantage because it helps us to keep our processors busy while they wait on asynchronous calls.

Let's consider our coffee shop again as an example. There are many items on the menu. However, it does not make sense to have a separate station for each item on the menu. Many stations would sit idle, much of the time. Instead, a single station can prepare multiple items, provided they all follow a similar processing flow.

The same holds true for our autonomous services. Each service is responsible to a single actor, but each needs to react to multiple event types in support of its functionality. Yet, it is not practical to maintain a separate channel and function per event type. So, we need a way to maintain good clean separation while processing multiple event types in the same stream processor.

Pipelines provide us with the mechanism to achieve this clean separation by dividing the processing logic into separate flows. We *fork* the master stream

into many pipelines and merge the pipelines at the tail end. Each pipeline defines a set of related steps that apply to a specific event type and content.

The following code block shows our original stream processor, from the *Processing event streams* section, implemented as a pipeline:

```
const pipeline1 = (options) => // initialize (stream) => stream // assemble .filter(onEventType) .map(toUpdateRequest) .through(update({ parallel: 4, ...options })); ... export default pipeline1;
```

A pipeline is a curried function with the following functions:

- The first function call will initialize the pipeline with the `options` for configuration.
- The second call receives a forked `stream`, so we can assemble the steps of the pipeline.

A pipeline typically starts with one or more filter steps that dictate which event types apply to the pipeline and will flow through the remaining steps. We implement each pipeline in its own module file to create clean separation, and so we can test them independently. Note that we have further decoupled the pipeline from the channel technology decision because we call the `fromKinesis` function, which we introduced in the *Creating a stream* section, when we assemble the pipelines, as we will see next.

Now, we need to assemble a set of pipelines into a stream processor. The following code block shows our original `listener` function, from the *Processing event streams* section, implemented using pipelines:

```
import { initialize, defaultOptions, ... } from 'aws-lambda-stream'; import pipeline1 from './pipeline1'; import pipeline2 from './pipeline2'; const PIPELINES = { pipeline1, pipeline2 }; const OPTIONS = { ...defaultOptions, ... }; export const handler =
```

```
async (event) => initialize(PIPELINES, OPTIONS)
.assemble(fromKinesis(event)) .through(toPromise);
```

The `handler` code for a `listener` or `trigger` function is now mostly boilerplate. Here are the steps it takes to prepare a function to process a batch of events:

1. First, we import the pipelines and group them in the `PIPELINES` object.
2. Then, we initialize the pipelines with any `OPTIONS` instance.
3. Then, we create the master stream with the appropriate `from` function, such as `fromKinesis`.
4. Next, we assemble all the pipelines. The `assemble` function forks a stream for each pipeline and passes it to the curried function so that the pipeline can assemble its steps. At the tail end, the assemble function merges all the streams and adds the fault-handling steps that we covered in the *Designing for failure* section.
5. Finally, we use the `toPromise` function to start the flow of events through the processing pipelines.

From here, the pipelines take control and perform their work. If one pipeline is waiting on asynchronous calls, then the other pipelines will continue to pull work until they need to wait. This cooperative programming approach helps us to minimize the amount of time a stream processor spends doing nothing but waiting.

The processing concludes when all the units of work have flowed through all the pipelines.

You can find the library for the stream processing framework here: https://github.com/jgilbert01/aws-lambda-stream.

Now, let's create reusable pipelines.

Pipeline patterns

It is important to assemble a cohesive set of pipelines into a single function. Throughout the remaining chapters, we will see how to use pipelines in autonomous services to create `listener` and `trigger` functions for BFF, ESG, and Control services.

As we build autonomous services, it becomes clear that we have repeatable pipeline patterns that we use over and over again, with consistent differences. For example, a `listener` function in a BFF service is responsible for consuming domain events and creating the materialized views used in queries. All the pipelines follow the same steps. They filter for specific event types, map the data to an update request, and save the data in the data store. Only the details of the filter and map steps vary between the pipelines.

We can package these patterns into reusable pipelines and configure them with rules (that is, options). The following code block shows our original stream processor, from the *Processing event streams* section, implemented with a reusable pipeline pattern:

```
import { materialize } from 'aws-lambda-stream'; import {
toThingUpdateRequest } from './model'; const RULES = [{ id: 'p1',
pattern: materialize, eventType: /thing-*/, toUpdateRequest:
toThingUpdateRequest, }, { id: 'p2', pattern: materialize,
eventType: 'something-else', toUpdateRequest: (uow) => ({ ...uow,
... }), }]; export default RULES;
```

In this code block, we define a set of rules that we will use to initialize multiple reusable pipelines. Each rule declares a set of properties:

- The `id` field contains a unique name for the pipeline instance.
- The `pattern` field points to the function that implements the pipeline pattern. In this case, we use the `materialize` pipeline function provided

by the library.

- The `eventType` field contains a regex, string, or array of strings used to filter on the event type.
- The `toUpdateRequest` field is specific to the `materialize` pattern. This field points to the mapping function that the pipeline will call in the mapping step to format a DynamoDB update item request. You can define the function inline, but it is best to implement and unit test them separately.

Each rule requires an `id`, `pattern`, and `eventType` field, and all remaining fields are specific to the pattern. We can think of the rules as pipeline-specific configuration options. We merge each rule with the global configuration options when we initialize the pipeline.

The following code block shows how we change the `listener` function to initialize the pipelines from the rules:

```
import { initializeFrom, ... } from 'aws-lambda-stream'; ...
import RULES from './rules; const PIPELINES = {
...initializeFrom(RULES), pipeline2, };
```

The `initializeFrom` function iterates over the rules and prepares a pipeline instance for each, using the `pipeline` function specified in the `pattern` field. Note that we can initialize one-off pipelines alongside rule-driven pipelines.

In addition to the `materialize` pattern, the stream processing framework includes a `cdc` pattern that we will use in *Chapter 5, Turning the Cloud into the Database*, to create `trigger` functions. In *Chapter 8, Reacting to Events with More Events*, we will use the `collect`, `correlate`, and `evaluate` pipeline patterns to implement control services. You will find pipeline patterns in your own system, and you can package them into shared libraries.

Now, let's add parallel instances of stream processors.

Sharding

We introduced *Little's Law* in the *Living in an eventually consistent world* section. Little's Law shows us that work spends more time in a system as the system nears full capacity. Ultimately, the system will fall behind when the arrival rate exceeds the service rate. The law also shows us that parallelism has a significant impact on the amount of time spent in the system.

This is where sharding comes into play with stream processing. Sharding essentially divides one input channel (that is, stream) into multiple output channels (that is, shards). A separate instance of a stream processor consumes from each shard, which increases the parallelism and throughput of the system.

The `partitionKey` field in the event envelope plays an important role in sharding. A hashing algorithm on the partition key value determines which shard an event will flow through. The distribution of the partition keys dictates how well we utilize all the shards. For example, if we were to use the date for the partition key, then, on any given day, all events would flow through a single shard. We commonly referred to this problem as a **hot shard**.

We should lean toward using a domain entity identifier for the partition key. These identifiers usually contain a V4 random UUID, which helps to ensure an even distribution across the shards. This also ensures that related events for a domain entity instance will flow through the same shard.

Now, let's see how we can group these related events to gain efficiencies and further improve throughput.

Batching and grouping

Batching and grouping are similar because they reduce the number of calls a processor makes to the target resource. This reduces the amount of time a processor spends waiting on network latency and it also reduces the amount of work the target needs to perform. In the case of grouping, it can significantly reduce the load on the target.

In the *Processing event streams* section, we introduced the concept of a **unit of work**. A unit of work should succeed or fail together. In the *Designing for failure* section, we also attached the unit of work to a *fault* event so that we could resubmit the unit of work. In general, each incoming event represents one unit of work.

Now, we are looking to optimize throughput by collecting multiple events into a single unit of work. This is where we make use of the optional `batch` field that we specified in the `UnitOfWork` interface. When we batch and group, we gather events into a larger unit of work. We retain the individual units in this `batch` field to support fault handling and resubmitting all the individual events.

Batching

Before we can implement batching, we first need to determine whether the target resource supports it. Many services, such as Amazon Kinesis and DynamoDB, provide both a batch and a non-batch interface. Next, we need to determine the limits on the batch size.

From here, it is straightforward to implement batching in functional reactive programming. We add a step that collects units of work. When it reaches the limit, then the step makes the batch available for the next step to perform the call to the target resource.

The following code block creates a batch with a maximum size of 10:

```
import { toBatchUow, ... } from 'aws-lambda-stream'; export const
handler = async (event) => ... .batch(10) .map(toBatchUow)
.map(makeSomeAsyncCall) .parallel(4) ...
```

The Highland.js `batch` feature collects the batch, and the `toBatchUow` utility function formats the batch unit of work so that we can easily raise a fault for the entire batch. We can combine batching with the `parallel` feature to help ensure that the processor does not wait on these calls.

It is important to determine whether or not the API treats the batch as a single transaction or whether the individual entries in the batch can fail independently. If they can fail independently, then we need additional logic to resubmit the failed entries. This logic can be tedious, so it may be best to determine whether batching delivers sufficient improvements first.

Grouping

Multiplexing different types of events through a channel allows us to group related events together and reduce the number of calls we make against the target resource. For example, we could group events based on a common property, such as an account number, and just use the last event as input to the call.

The following code block creates a group based on the partition key:

```
import { toGroupUow, ... } from 'aws-lambda-stream'; export const
handler = async (event) => ... .group(uow =>
uow.event.partitionKey) .flatMap(toGroupUows)
.map(makeSomeAsyncCall) .parallel(4) ...
```

The Highland.js `group` feature reduces data into groups based on a field value, and the `toGroupUows` utility function formats each group as a unit of work so that we can easily raise a fault for the entire group. Again, we can

combine grouping with the `parallel` feature to help ensure that the processor does not wait on these calls.

We now have a set of techniques that help us optimize the throughput of our stream processors, ensuring that they don't fall behind, waste resources, or perform unnecessary work. In *Chapter 12*, *Optimizing Observability*, we will see how we can observe the throughput of our stream processors so that we can identify when optimization is necessary.

Summary

In this chapter, we learned that modern systems need to work in an eventually consistent world and that eventual consistency is based on transparency, facts, and a well-crafted chain reaction of atomic actions. You learned how to increase team confidence by using an event hub to create an outbound bulkhead between upstream and downstream services, and how the event sourcing pattern turns events into facts by saving them in perpetuity in an event lake.

We dug into the details, and you learned how to implement fault-tolerant stream processors and optimize a subsystem's messaging channel topology. Also, you learned how to optimize throughput with techniques such as multiplexing, pipelining, asynchronous non-blocking I/O, and sharding.

In the next chapter, we will cover the **Command Query Responsibility Segregation** (**CQRS**) pattern and see how it helps us fight data gravity, by implementing a data life cycle architecture that turns the database inside out and creates inbound bulkheads between downstream and upstream services. Plus, we will look at techniques to create idempotent and order-tolerant stream processors, how to create lean and high-performance data models, and how we can leverage CDC.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below: https://discord.gg/B8TubSxc35

5

Turning the Cloud into the Database

In *Chapter 4, Trusting Facts and Eventual Consistency*, we covered the event hub and event sourcing patterns and learned how they create an outbound bulkhead that protects upstream services from downstream outages. Now, we turn our attention to data architecture and how to reshape it to create inbound bulkheads that protect downstream services from upstream outages. Together, these bulkheads fortify the boundaries of autonomous services and give teams the confidence to forge ahead with changes, knowing that the boundaries will help control the blast radius when things go wrong.

In this chapter, we're going to cover the following main topics:

- Fighting data gravity
- Embracing the data life cycle
- Turning the database inside out
- Dissecting the CQRS pattern
- Keeping data lean
- Implementing idempotence and order tolerance
- Modeling data for operational performance
- Leveraging change data capture

Fighting data gravity

In *Chapter 1, Architecting for Innovation*, we saw that the role of architecture is to enable change so that teams can continuously experiment

and discover the best solutions for their users. When we think about architecture, we tend to focus on how we organize and arrange the source code to keep it clean and acyclic. However, a system's data is arguably its most valuable asset and simultaneously its biggest barrier to change. This is because data has gravity. As a system grows and evolves, so too does its data and its data structure. We must spend as much, or more, effort on how we organize and arrange our data as we do on our source code. Otherwise, it will become brittle, inflexible, and impossible to maintain.

When these deficiencies are combined with the sheer volume of data, the weight of our data becomes an intractable force that prevents teams from moving forward. This impact on a system's ability to grow and evolve is a measure of the data's gravity. To fight data gravity and ultimately escape its pull, we must first understand the forces that so negatively impact our ability to change a system's data architecture.

Let's look at how competing demands, insufficient capacity, and intractable volumes influence a system's data gravity.

Competing demands

In *Chapter 3, Taming the Presentation Tier,* we saw how the presentation tier has acted like a pendulum, oscillating back and forth between client-side and server-side rendering. There is a pendulum in motion at the data tier as well, but it is moving much more slowly between granular and monolithic datastores. In *Chapter 1, Architecting for Innovation*, we saw how the isolation of information silos drove our industry toward monolithic architecture. However, monolithic databases have proven to be equally deficient, because the force of their data gravity impedes innovation.

In *Chapter 2, Defining Boundaries and Letting Go*, we found that people (that is, actors) are the driving force behind change. The **Single Responsibility Principle** (**SRP**) tells us that each module must be responsible to one and only one actor. However, we tend to ignore this principle at the data tier in monolithic databases. In fact, we encourage the reuse of database tables.

This is similar to the topic of whether the presentation tier should live on the client side or the server side. It takes a lot of discipline not to couple business logic with presentation logic when the code resides in the same repository. The same holds true for monolithic databases because all the tables are easily accessible by all modules using the same database.

Furthermore, relation data modeling teaches us to normalize database schemas to the third degree (that is, the third normal form) so that there is no duplicate data. This has helped foster the belief that all duplication of data is bad. In other words, it incentivizes the sharing of tables between modules.

Each module that accesses a shared table likely represents a different actor; each places competing demands on the structure of the database schema. Over time, the schema grows and contorts to accommodate the different requirements. This results in hidden dependencies between the modules. A misunderstanding of these hidden dependencies can lead to mistakes that cause system outages.

Ultimately, these competing demands reduce team confidence, and the force of data gravity starts to grow as we resist making changes.

Insufficient capacity

It’s a classic story. You put an application into production, and it performs great for months. Then, one day, it just slows down. After wracking your brain for days, you learn that your application is no longer the sole owner of the database infrastructure. There are now a dozen or so applications competing for the same limited database resources.

This scenario is typical, because owning and operating a database, with all the safeguards in place, is not easy. Our data is our most important asset, and we must protect it. However, hardening a database for security and redundancy takes skill, and database administrators are overworked. So, there is a disincentive for running more and more databases. Instead, we make a single database support more and more applications and we add more and more vertical and horizontal capacity. It never seems to be enough. There is always insufficient capacity, so we keep adding more and more.

But now we have created a monstrosity. The database infrastructure has become incredibly complex. Only a select few database administrators are entrusted with making significant changes. Redundancy is baked into the system, but making changes is perceived as high risk because the wrong mistake could cause an outage for all the applications. So, we resist change and the force of data gravity grows stronger.

Intractable volumes

Having lots of data is generally a good thing. The problem is when and where we use it. Our monolithic databases tend to collect data and never let it go, just in case we need it. Meanwhile, the applications that create and use the data need to evolve to support new requirements. This inevitably impacts the data schemas, so the old data must be converted. As the volume of data grows, the conversion effort becomes more difficult and time consuming.

But the applications no longer use much of this data, so why bother converting it? More than likely, the competing demands for the data have created so many interdependencies that we cannot archive the old data. So, it has to stay where it is and as the data grows, it becomes more and more difficult to govern and manipulate. It becomes intractable. It's a compounding effect. We can't archive it, so we must add more capacity. It's difficult to convert, so we resist change, and the force of data gravity grows ever stronger.

The bottom line is that monolithic databases have no bulkheads. There is no real data autonomy. Sooner or later, data's gravity will cause a system to implode or, worse yet, cause innovation to grind to a halt. We need to escape data's gravity by defining fortified boundaries and splitting up the data.

This is what cloud-native is really all about: leveraging the power of the cloud and turning the cloud into the database. Data life cycle architecture and turning the database inside out are two complimentary approaches to breaking up monolithic databases. Let's start by looking at how we use data over its life cycle.

Embracing the data life cycle

We need to break up monolithic databases to ensure that our data's gravity does not impede innovation. A natural inclination is to divide a database into sets of related tables (that is, nouns) and wrap each of these new databases with a data access service. However, this approach is an anti-pattern, as it does not address the problem of competing demands. It just moves data gravity to a service layer and breaks one big problem into multiple smaller problems. It will delay the onset of the force of data gravity, but each database will eventually succumb. We need a better approach.

In *Chapter 2, Defining Boundaries and Letting Go*, we discussed **event-first** thinking along with the advice that we should **focus on verbs instead of nouns**. Plus, we covered multiple approaches for discovering the boundaries between autonomous subsystems and we introduced a set of autonomous service patterns for decomposing a subsystem into services. For breaking up monolithic databases, the most applicable of these approaches and patterns are the **data life cycle architecture** and the **Backend for Frontend** (**BFF**) pattern.

Data Life Cycle Architecture (**DLC**) recognizes that over the life of a piece of data, different actors interact with the data at different phases in its life cycle. These actors are the sources of the competing demands for data. Following the SRP, DLC recommends breaking the database up along the phases of the data life cycle. This approach entails dividing a database along an entirely different dimension than you would expect, but it is not unprecedented. This is akin to the traditional approach of separating an **Online Transaction Processing** (**OLTP**) database from an **Online Analytical Processing** (**OLAP**) database. We are just applying this approach at a much more granular level.

In addition to functional differences, these actors have different access patterns and storage requirements. This is the perfect opportunity to apply the **polyglot persistence** practice by picking the right database technology for each phase of the data life cycle. We will see which database technologies fit the different phases best.

In *Chapter 2, Defining Boundaries and Letting Go*, we used the event storming technique to help us discover the events (that is, verbs) that a system will emit, the relative order of those events, and the actors who produce them. During this process, we also identify the domain aggregate

(that is, the data) that is in play when the events are produced. Using this technique, we naturally discover the phases of the data's life cycle, because it identifies the order in which the different actors interact with the domain aggregates.

Generally speaking, we will implement a BFF service for each actor, domain aggregate, and life cycle phase combination. Each of these autonomous services will own its slice in time of the data and use the database technology that best suits the access patterns:

Figure 5.1: Data life cycle phases

As depicted in *Figure 5.1*, we can classify the data life cycle into four phases: create, use, analyze, and archive. Let's look at each of these phases in turn.

Create phase

We are surrounded by and interact with data on a daily basis. This data originates from a variety of sources. We create much of the data manually, but a lot of it is generated, and from a system's perspective, data is often just received from an external source. There are many examples of data creation in our hypothetical food delivery system.

Restaurants create their menus manually and the delivery system receives the menu data through an ingress interface. Customers browse the menus, and the delivery system generates click-stream data to track customer activity. Customers manually create orders, and the delivery system sends orders to restaurants through an egress interface. The delivery system

generates delivery assignments and the drivers manually accept the assignments. Restaurants generate order status data as they fulfill orders and drivers generate delivery status data as they carry out a delivery.

In the create phase of the data life cycle, we want to use datastores and data formats that are optimized for writing. For generated and integrated data, we can simply write events to a stream, such as AWS Kinesis. This is the stream-first variation of the Event Sourcing pattern that we covered in *Chapter 4, Trusting Facts and Eventual Consistency*.

For manual data entry, we should use a normalized data format, so that users do not need to enter duplicate data. Key-value, document, and relational databases, such as *AWS DynamoDB* and *AWS Aurora*, are well suited for data entry. It is important that the selected database provides a **Change Data Capture** (**CDC**) feature so that we can implement the database-first variation of the Event Sourcing pattern. We will cover this shortly in the *Leveraging change data capture* section.

Another important characteristic of data creation is the rate at which we create it. We create some data infrequently, such as reference data or master data, whereas we produce transactional data continuously. We can refer to these as slow data and fast data, respectively. We will cover the implications of slow data in the *Soft deletes* section. We will see how to control the volume of fast data in the *Keeping data lean* section.

Upstream services create data and produce domain events so that downstream services can react to the data in the use phase of the data life cycle. Let’s move on to the use phase.

Use phase

We create all data for a purpose. But the actors who create the data and those who use it are almost always different. The different actors have different requirements, so we want to separate these concerns into different autonomous services. We will let upstream services create the data and let an unlimited number of downstream services use the data in a format that best suits their needs.

For example, in our hypothetical food delivery system, restaurants create their menus and customers use the menus of many restaurants. However, the access patterns and performance requirements are completely different. The way customers create orders and how drivers and restaurants use orders are very different. In fact, the way drivers and restaurants use orders are very different from each other, and the way customers track the status of their current orders and view the history of their previous orders are each different access patterns.

We traditionally refer to the databases that we use for these scenarios as OLTP databases or operational datastores. It is important to note that their access patterns are not ad hoc. We explicitly define these datastores to allow the actors to perform their tasks efficiently. We want to take full advantage of this characteristic and optimize these operational datastores for their specific scenarios. In the *Dissecting the CQRS pattern* section, we will see how to create materialized views that prejoin data for fast access, and in the *Keeping data lean* section, we will cover techniques for trimming the fat from these databases.

Key-value stores, search engines, and object storage are a great fit for these operational datastores. Examples of NoSQL services in these categories include AWS DynamoDB, Elasticsearch, and AWS S3. We will see examples of DynamoDB in the *Modeling data for operational performance* section, and we will see an example of using Elasticsearch in combination with S3 in

Chapter 6, A Best Friend for the Frontend. These usage scenarios can also benefit from caching at the edge with CDN services such as AWS CloudFront. In very high-performance scenarios, a durable cache such as AWS **DynamoDB Accelerator** (**DAX**) may be warranted.

The create and use phases of the data life cycle are very interrelated because each autonomous service can use data created by upstream services and create more data for use in downstream services. However, it is very important to draw a line between these phases and the analyze phase because analytics access patterns are very different from operational access patterns. Let's move on to the analyze phase.

Analyze phase

All upstream data creation leads to the analyze phase. In *Chapter 2, Defining Boundaries and Letting Go*, we discussed how event-first systems implicitly build business analytics and observability into a system because events represent facts, and analytics deals in facts. There are many potential uses of analytics in our hypothetical food delivery system.

The delivery system tracks customer activity so that it can better target customers and drive up sales conversion. Restaurants need statistics so that they can gauge whether or not their partnership with the delivery service is producing positive results. Delivery statistics can help identify drivers that need more training. We can derive all this actionable information from the events generated by the system.

All the traditional analytics techniques and technologies are still applicable. For example, data marts and data warehouses can consume upstream events and use them to populate the fact tables in their star schemas. We leverage columnar databases for this purpose, such as AWS Redshift and AWS

Athena. These are relational databases that store data on disk organized by column instead of by row.

This is because the large amount of data makes it impractical to create indices on tables. Storing data by column effectively creates an index per column. Time series databases, such as InfluxDB and AWS Timestream, and search engines, such as Elasticsearch, are alternatives to using relational databases for analytics.

One of the most significant advancements in the analyze phase of the data life cycle is the event lake. The event lake consumes all events and stores these facts in perpetuity. This allows us to defer commitment on how we want to utilize analytics technology. For example, teams can experiment with different hypotheses by replaying historical events from the lake to a datastore and validating whether or not the approach produces accurate forecasts. It also supports creating data marts on demand and tearing them down afterward to reduce cost. We covered event lakes in *Chapter 4, Trusting Facts and Eventual Consistency*.

Another significant advancement is in actionable analytics with big data and machine learning. Services consume events of interest and use them to train machine learning models. The models are then *used* by downstream services to identify conditions, such as in anomaly detection, and control the behavior of the system. We will cover additional details in the *Leveraging ML for control flow* section of *Chapter 8, Reacting to Events with More Events*.

Data mesh is an emerging topic in the analyze phase. Its main goal is to eliminate the bottleneck of overworked, centralized data science and analytics teams, by decentralizing and federating the activities of preparing analytics data and making is readily available. Our concepts of autonomous subsystems and external domain events are well aligned with data mesh

because a subsystem's external domain events are sanitized and purpose made for external consumption.

Ultimately, the analyze phase plays an important role in driving innovation because it provides the insights that help organizations learn and adapt. However, it is important to recognize that analytics services support different demands and need their own datastores.

Now, let's move on to the archive phase.

Archive phase

Sooner or later, the cost of keeping a piece of data readily available will become more expensive than the value it provides. As we saw in the *Intractable volumes* section, keeping data too long increases the force of data gravity, which inevitably impedes innovation. The solution is to archive data and keep operational data lean, but these are separate concerns that we need to handle independently to avoid adding complexity.

In the *Keeping data lean* section, we cover the time-to-live technique. This is a simple technique where each service decides how long it needs to use a piece of data. After that time, the database deletes the row so that the datastore can maintain a predictable and manageable size. These services do not concern themselves with the requirements of archiving because archiving requirements apply to different actors.

Instead, we create downstream archiving services that consume events and **proactively archive data**. In other words, we create the archives continuously as we create the data. This frees the operational services to delete data at will. The archiving services handle the records management requirements regarding what data we need to retain and for how long. We will see an example of this in *Chapter 6*, *A Best Friend for the Frontend*.

Object storage, such as AWS S3, is a great fit for archiving. As the data ages, it can transition to long-term storage, such as AWS Glacier. We typically organize these archives by actor and date. In our hypothetical food delivery system, we would organize the archive around restaurants, drivers, and customers. The archive would collect all the events and any documents.

An archive will likely support re-hydrating data so that upstream services can reconstitute the data they have deleted. For example, in a case management system, we could reopen a closed case. An upstream service could produce a **case-reopened** domain event and the archive service would produce **re-sync** events once it retrieves the cold data, similar to the approach discussed in the *Soft deletes* section.

Creating a data architecture that decomposes a monolithic database along the phases of the data life cycle addresses the problem of competing demands. However, there is an ever-present concern about the amount of duplicate data the approach creates and the ability to keep it synchronized. It’s an understandable concern, but there is relatively little duplication in comparison to what traditional databases actually do. Let’s look at what a database does inside so that we can turn the database inside out and ultimately turn the cloud into the database.

Turning the database inside out

When I built my first cloud-native e-commerce system, I knew I had to break out of the relational database mold. It was reasonable to expect that I could use AWS RDS MySQL for the back-office users who would maintain the product catalog, but I would need something different to support the

expected volume of customers reading from the catalog. I was already using Amazon S3 and Amazon CloudFront to serve SPA applications, with no concern for supporting any load volume. So, why couldn't I do the same thing for the catalog data?

The catalog data was public, and it was static. Offers didn't change; they just expired and were replaced by new offers. I could put the images and offer JSON data into an S3 bucket with a path structure that supported good RESTful URLs, such as `/merchants/123/offers/987/images/abc`, and set the **cache-control** property on the objects so that CloudFront would serve the requests more often than not. I combined this with Amazon CloudSearch to index the catalog and I had a solution that I was confident would scale with the user base and would be reasonable to operate.

The only remaining piece of the puzzle was synchronizing the catalog data from RDS to S3 and CloudSearch. Thanks to my experience with **Enterprise Application Integration** (**EAI**), this was already a familiar problem. I just had to select the right cloud-native tools. Amazon Kinesis and AWS Lambda proved to be a perfect fit.

I didn't realize it at the time, but I was implementing the Event Sourcing and CQRS patterns. Along the way, I came across Martin Kleppmann's excellent article, *Turning the database inside out with Apache Samza* (https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza). This article put a fine point on what I was doing and helped solidify the approach for breaking away from monolithic databases, maximizing scale and flexibility, and keeping all the pieces working in harmony.

This last point is the real crux of the matter. We are accustomed to relying on the database to do a lot of the heavy lifting for us, but we no longer have one

homogeneous database at the core of our systems. We now have many, heterogeneous databases, and it is incumbent on us to make them all work together. This, understandably, seems like a daunting challenge. But, other than query optimization, databases really aren't that magical.

When we think of a database, the first thing we typically think about is tables. After that, we think about executing queries against those tables. But this barely scratches the surface of everything a database is actually doing for us under the hood. Yet, when we look inside, it turns out that the thing we are most concerned about, *data synchronization*, is not that daunting. We must overcome the concern of taking over this responsibility from the database because taking control of *data synchronization* is what leads to fully escaping data's gravity. So, let's address this concern by looking inside to see how databases use a transaction log to manage (that is, synchronize) derived data. Then we will use this knowledge to move our data through its life cycle.

The transaction log

At the heart of a database is the transaction log. It is the source of truth for information about every insert, update, and delete statement that has been executed against a database.

For example, when we execute an update statement against a specific table, the database finds the desired record, applies the changes, and then stores the previous state and the new state in the transaction log. In other words, the transaction log records all the state-change events that occur within the database.

However, our modern systems now have many databases, which means that we have many transaction logs, but none of them provides a definitive

source of truth about the state of the entire system. The solution is the event hub and event lake, as we covered in *Chapter 4, Trusting Facts and Eventual Consistency*. This is exactly what they accomplish. They provide the source of truth for information about every command that the autonomous services have performed within the system. Now, let's look deeper into what a database does with its transaction log, so we can understand what we can do with a system-wide stream of events.

Derived data

The transaction log is the heart of the database because it is used to push data out to the other parts of the database. These other parts of the database are the derived **copies** of the data, such as indices, materialized views, and replicated instances of the database. It's easy to overlook just how much-derived data there is in a typical database because the database manages and synchronizes these copies for us. But all this work the database is doing can have a steep cost.

The database uses the log to replicate the database to other database instances for failover and to create more copies in the form of read replicas that support more and more readers. It maintains indices, which are copies of the data arranged on disk in different sort orders to support faster reads. It calculates materialized views, and each is yet another copy of the data that contains the precalculated results of a query that is too expensive to execute over and over again in real time.

All of this work is not too much of a burden for a small, focused database with a predictable load. However, as we add more and more tables, indices, and materialized views to the database, and more and more consumers of the

data, then all this competition for scarce resources adds up and drives down performance for end users.

The inevitable consequence is that applications add their own read-through cache to shield their users from this competition with other applications. This is yet another derived copy of the data. However, this time the copy is not managed by the database but by the application developers. The developers have to map the data to the desired format, handle cache misses, and account for stale reads. This additional code and infrastructure adds to the complexity and cost of the applications, and the pull of the data's gravity continues to grow.

This is the way we have built applications for decades. But what if we stop and instead turn the database inside out? What if we create a unified transaction log across the databases? What if we take all the indices and materialized views and move them into the datastores of the services that use them? What if we can eliminate the need for applications to maintain their own read-through cache? What if we can stop centrally managing a one-size-fits-all read replication strategy? What if we can eliminate the single point of failure within the system and build bulkheads between the services? What if we could choose a different database technology on a query-by-query basis? What if we could spread the load of all this processing across the cloud?

What would this look like? We already have an event hub and event lake, which act as the system-wide transaction log. Now let's take a look at how a system-wide view of the CQRS pattern enables us to turn the cloud into the database.

Dissecting the CQRS pattern

The **Command Query Responsibility Segregation** (**CQRS**) pattern provides the foundation for turning our monolithic databases inside out and escaping data's gravity. It proposes the use of two different domain models, a write-optimized command model and a read-optimized query model. Separating these two concerns allows each to change and evolve independently and enables the creation of systems that are more responsive, resilient, and elastic.

Unfortunately, the CQRS pattern has a reputation for making systems more complex. This concern is not unfounded, however; with the right alterations, this segregation produces systems that are actually more straightforward and flexible. A few examples will help make this clear. Let's look at an example without CQRS, one with what I refer to as traditional CQRS, and a third example with the altered form that I refer to as system-wide CQRS.

Figure 5.2 depicts a BFF service that provides read and write capabilities to the end user without using separate domain models:

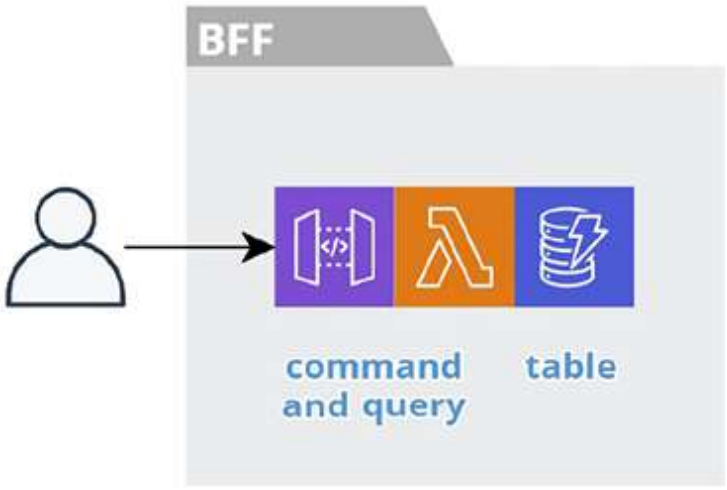


Figure 5.2: Without CQRS

This results in a straightforward development experience for the vast majority of use cases, which only have **Create, Read, Update, and Delete**

(**CRUD**) requirements. We can easily implement this with a single table and a single RESTful command and query service using **HTTP PUT**, **GET**, and **DELETE** methods.

Figure 5.3 depicts a BFF service that provides read and write capabilities to the end user using separate domain models:

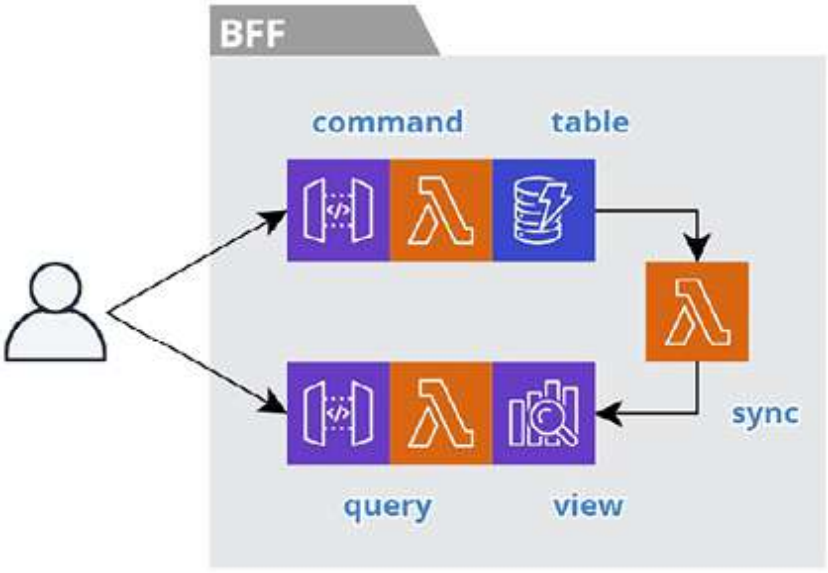


Figure 5.3: Traditional CQRS

This is obviously more difficult to implement because there are more pieces. There is a **table** used by the **command** function, a **view** used by the **query** function, and a **sync** function to synchronize the data from the table to the view. If there is a distinct benefit to separating the domain models, such as using polyglot persistence, then it may be worth spending the extra effort. For example, in this case, we implement the view with a search engine that can support more expressive queries.

Now, let’s look at how we can improve CQRS.

System wide CQRS

One important thing to notice in these first two examples is that they each focus on supporting the usage scenarios of a single actor. This is great from the perspective of the SRP. However, we also need to consider the system as a whole.

Going back to the *Embracing data life cycle* section, we saw that over the course of the data life cycle, different actors use the same logical data with different access and storage requirements. We proposed using the data life cycle phases to define the boundaries between different datastores. In the *Turning the database inside out* section, we saw how these competing demands result in the creation of derived data, such as materialized views and indices, and proposed moving them within the boundaries of the services that use them. As a consequence, the CQRS pattern must take on a much more strategic role when we adopt this holistic perspective.

Figure 5.4 depicts two BFF services collaborating to support multiple actors in different life cycle phases:

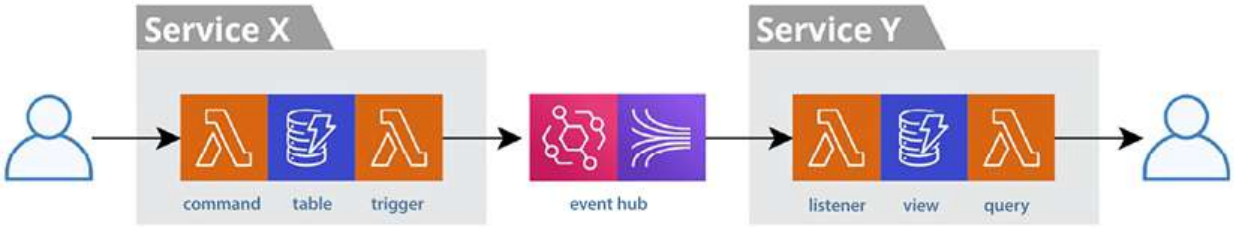


Figure 5.4: System-wide CQRS

We saw a similar version of this diagram in *Chapter 1*, *Architecting for Innovation*, when we discussed the **Command**, **Publish**, **Consume**, **Query** (**CPCQ**) flow.

Service X sits upstream, supporting the first actor, and provides a **command** function that *creates* data in a **table** that it owns. The **trigger** function produces a domain event to the **event hub**, which provides the system-wide

transaction log that we discussed in the *Turning the database inside out* section. We will discuss this trigger mechanism in detail later, in the *Leveraging change data capture* section.

Service Y sits downstream, supporting the next actor, and provides a **listener** function that consumes events and materializes the needed data in a **view** that we specifically designed to support the access scenarios of this downstream actor. Furthermore, the **query** function and **view** provide the downstream actor with dedicated resources for *using* the data to help ensure a responsive and resilient user experience.

This last example shows how the CQRS pattern is a key mechanism for creating an architecture that enables change. We employ it to create derived data that supports one and only one actor so that we can turn our monolithic databases inside out and fight data's gravity. Along with these materialized views come the benefits of inbound bulkheads, live cache, and capacity allocation per query. Let's dig into each of these, starting with a look at how to create materialized views.

Materialized views

The primary job of a materialized view is to improve the responsiveness of the user experience. We accomplish this by choosing the most appropriate type of database, creating proper indices, and then pre-calculating joins and storing the data in the format needed by the specific consumer. There is no reason to calculate the join over and over again. We can do it once when we receive the data so that the consumer does not pay the price on every request. This also reduces the cost of our serverless solution, since the price is based on execution time.

We have already seen how upstream services produce events, in *Chapter 4*, *Trusting Facts and Eventual Consistency*, using the stream-first variation of the Event Sourcing pattern, and we will look into the database-first variation of the pattern shortly in the *Leveraging change data capture* section. Now, let's look at how downstream services implement the CQRS pattern to create their materialized views.

The following stream processing example builds on the concept of pipeline patterns, which we covered in *Chapter 4*, *Trusting Facts and Eventual Consistency*:

```
import { materialize, updateExpression } from 'aws-lambda-stream
const RULES = [{
  id: 'm1',
  pattern: materialize,
  eventType: /thing-(created|updated)/,
  toUpdateRequest: (uow) => ({
    Key: {
      pk: uow.event.thing.id,
      sk: 'thing',
    },
    ...updateExpression({
      name: uow.event.thing.name,
      description: uow.event.thing.description,
    }),
  }),
}, {
  id: 'm2',
  pattern: materialize,
  eventType: 'thing-submitted',
  toUpdateRequest: (uow) => ({
    Key: {
      pk: uow.event.thing.id,
      sk: 'thing',
    },
    ...updateExpression({
      status: uow.event.thing.status,
    }),
```

```
  }),
}];
```

This example is joining the data from three different types of domain events into a single record in the datastore. Here are the typical steps for creating a materialized view:

1. Filter for the desired events.
2. Map the data in the event to the needed format.
3. Save the data to the datastore.

These steps are so typical that it makes sense to wrap them in a reusable pipeline pattern. This example uses a **materialize** pipeline pattern that works with Amazon DynamoDB. Similar pipeline patterns are available for Amazon S3 and Elasticsearch. And of course, you can create your own or implement CQRS in your favorite language and framework.

NOTE

The mapping code in this example is included inline for brevity. These functions will normally be implemented and tested in separate modules.

The most interesting details are in the `toUpdateRequest` functions that implement the mapping step. We are joining the data from the different events based on the identifier (that is, `uow.event.thing.id`) of the `Thing` domain entity. Two separate rules are defined because the thing-created and thing-updated event types carry the `name` and `description` fields, and the `thing-submitted` event type carries the `status` field.

These events can arrive in any order, so we are using the DynamoDB `UpdateItem` operation as opposed to the `PutItem` operation. Both operations will create a new record if one does not exist for the specified `Key` or update the record if it does exist. But the `PutItem` operation will replace all the fields, whereas the `UpdateItem` operation only replaces the specified fields, name and description, or status. This allows us to weave together the fields from the different domain events and incrementally create the desired join records. It also allows us to do this without retrieving data first, which improves throughput. We will go into more detail later in the *Implementing idempotence and order tolerance* section.

Inbound bulkheads

Materialized views also perform the equally important job of providing the system with resilience. As depicted in *Figure 5.5*, services that have direct dependencies on other services are less resilient because a disruption in the dependee service will impact the depender service. This results in an architecture that does not enable change, because teams become reluctant to make changes for fear of the ripple effect if they make a mistake.

To counteract this impulse, we must fortify the boundaries between services by adding bulkheads that control the blast radius when we inevitably make a mistake:

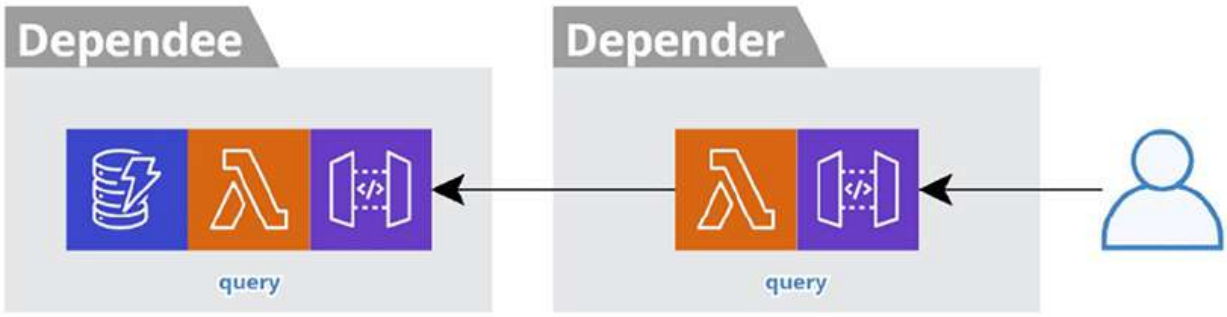


Figure 5.5: Synchronous communication anti-pattern

The Circuit Breaker pattern attempts to solve this problem while maintaining the synchronous communication paradigm. When the depender service detects that it is receiving too many errors from the dependee service, the depender will open a circuit and stop calling the dependee service for a period of time. But this approach really only helps the dependee service by reducing the load so that the dependee can have an opportunity to recover. Meanwhile, we force the depender service to return some form of default response to its end users.

This results in a suboptimal user experience when there is a disruption, adds overhead to every request even when there is no disruption, makes the developer experience more complicated, and increases the total cost of ownership. There are cases when the complexity of this approach is unavoidable, and we cover this in *Chapter 7*, *Bridging Intersystem Gaps*. But for all other scenarios, the best approach is to eliminate the dependencies by eliminating synchronous inter-service communication and leveraging materialized views.

In *Chapter 4*, *Trusting Facts and Eventual Consistency*, we learned how to implement asynchronous inter-service communication and use an event hub as an outbound bulkhead to protect upstream services from disruptions in downstream services. The event hub is serverless, so upstream services can rely on it to receive their events when they publish them. Downstream services will eventually consume these events and make themselves consistent.

The CQRS pattern is employed by downstream services to react to these events and create the materialized views that act as an inbound bulkhead to protect downstream services from disruptions in upstream services. Again,

downstream services leverage highly available, fully managed, serverless services to implement their dedicated datastores, so we can be confident that they will respond to our queries.

When an upstream service has an outage, there is no impact on downstream services because they can continue to respond to queries with the most recent data in their materialized views, instead of a default response. This also optimizes responsiveness, because there are no additional layers involved in these requests. It also reduces the complexity of the developer experience and lowers the total cost of ownership.

Ultimately, these bulkheads make services more than resilient; they make them autonomous. These materialized views are the resources services need to continue operating when other services are experiencing problems. This, in turn, gives teams the confidence to experiment and increase the pace of innovation, which is our primary architectural goal.

Live cache

Another benefit of the CQRS pattern is that materialized views act as a live cache, which solves the inherent problems of a read-through cache. As we previously discussed in the *Turning the database inside out* section, developers will ultimately turn to adding a caching layer to their technology stack to compensate for the increased latency that results from competition for scarce database resources. However, as we have seen, a read-through cache just addresses the symptoms instead of the root cause, and it introduces its own problems as well.

The first problem is cache misses. When a value is missing in the cache, we need to make an additional call back to the source database to retrieve the data, hence the name read-through. Then we add the data to the cache before

we return it. As a result of this additional work, the responsiveness of the user experience is inconsistent. The extreme case is when the cache is cold (that is, empty) and all requests are cache misses.

The next problem is stale data. When there is a cache hit, we return the data from the cache instead of from the source database. This is the purpose of a cache, but it introduces the possibility that we could return stale data if the data has changed in the source database. The workaround is to set a time to live that will expire entries from the cache to force cache misses. The trick is finding the right balance between the likelihood of returning stale data and having too many cache misses. If we set the value too low, then we will have more misses; if we set it too high, we will return stale data.

The final problem is complexity. A read-through cache adds another layer of technology to the call stack. The logic to handle a cache miss makes the developer experience more complex and increases the likelihood of introducing bugs. All in all, a read-through cache increases the total cost of ownership.

The CQRS pattern and materialized views provide a better solution because we keep the data up to date in near real time. This simplifies the developer experience because the code to materialize data is separated from the code to retrieve the data. The responsiveness of the user experience is consistent because there are no additional layers and calls. The use of a dedicated, highly available, serverless datastore, such as Amazon DynamoDB, removes the need for any additional layers because of their inherent high performance. Furthermore, we get to make this polyglot persistence decision on a query-by-query basis, which we will cover next.

Capacity per reader, per query

As we have seen, placing all data in the same database leads to competition for scarce resources and creates a single point of failure. Moving the data into separate databases and wrapping each with a service just moves these problems to another location because many services still share the data. The CQRS pattern provides a complete solution by giving each reader total control over its own derived data. We are, in essence, allocating capacity on a per-reader, per-query basis.

Each service supports a different reader (that is, actor) and creates and maintains its own materialized views and indices, instead of relying on a central, shared database. What's more, we can choose to use a different type of datastore for each query (that is, each materialized view). This gives each service complete autonomy to allocate the capacity it needs. Nothing is shared, so there is no competition for resources. If the responsiveness of a service degrades, then it is clear that the problem is self-inflicted.

This is a very powerful approach to fighting data gravity. We cannot accomplish this with a shared database, not even multiple shared databases. We are spreading the load across the cloud. In other words, we are not just turning the database inside out, we are ultimately turning the cloud into the database.

Up to this point, we have addressed the competing demands and insufficient capacity challenges that generate data gravity and impede innovation. By turning the database inside out and leveraging the CQRS pattern, we have given each service full control over its data. Now, we turn to the intractable volume challenge and address it by keeping data lean.

Keeping data lean

Intractable volumes of data, as we have already seen, are one of the causes of data gravity. The sheer volume of data in a system can impede its evolution because the time and effort involved in reshaping data is a powerful deterrent. In the *Embracing data life cycle* section, we discussed how defining boundaries between the data throughout the phases of the data's life cycle makes a big improvement as we move these groups of data into separate, leaner databases.

In the *Turning the database inside out* section, we saw that a large portion of a database's size is attributable to derived (that is, duplicate) data, such as indices and materialized views. Moving this derived data into the datastores of the services that use it makes the source datastores even more lean.

But we can do more. Upstream services produce events as they create data, and these events become the source of truth in the system-wide transaction log. This frees services to pick and choose the data they need to retain. Downstream services can project less data to keep their materialized views lean and all services can delete data they no longer need. Let's see how.

Projections

Duplication of data is one of the overarching concerns with the CQRS pattern. Setting aside the fact that traditional databases are full of duplicate data, we can use projections to help keep downstream datastores as lean as possible. Projections are a normal feature of databases. The most common example is the `SELECT` clause of an SQL statement, which defines the columns to project in the response.

Similarly, when we create materialized views from upstream events, we do not need to retain all the data that is in an event. The event lake acts as the system-wide transaction log and record of truth so that other services do not

have to. This means that downstream services can project the minimum amount of data into their views that is needed to support their end users. For example, a BFF service may just need the `id`, `name`, and `description` from a domain entity so that it can present a drop-down list to the user.

Keeping materialized views lean obviously helps to reduce storage costs, but it also improves performance and reduces overall costs. For example, a serverless database, such as Amazon DynamoDB, is priced based on the amount of capacity that is consumed per request. The amount of capacity a query consumes is dependent on the amount of data that it retrieves. Therefore, we do not want to process any more data than is necessary. In other words, retrieving data that we will not use is a waste of money. If we keep our data lean, by excluding unneeded data from our views in the first place, we can ensure that we are not wasting capacity.

Keeping data lean also improves the performance of queries, which reduces the time that functions take to execute and thus reduces costs further. This is one of the benefits of the serverless-first approach. Doing the right thing, such as keeping data lean, provides the double benefit of reducing costs while also improving the responsiveness of the user experience. This, in and of itself, makes the cost of any duplicated data money well spent.

Projections also help reduce the impact of changes in upstream services. In *Chapter 11, Choreographing Deployment and Delivery*, we cover the robustness principle, which states that modules should be conservative in what they send and liberal in what they receive. Following this principle, if an upstream service makes a change to a field that a downstream service does not include in its projection, then those changes can and should be safely ignored. Thus, projections have the positive effect of helping to minimize the dependencies between services.

Time to live

While projections are important for keeping individual records lean, we also need to address the total number of records. Too often we let tables grow and grow even though we only use the most recent data on a daily basis. We typically keep the older data around just in case we happen to need it in the future, but that is a different usage scenario. In the *Embracing data life cycle* section, we covered the concept of proactive archiving. The gist is that dedicated services take responsibility for archiving data in real time, so that other services can focus on just the data they need.

In the context of a specific service, data has a useful life on a query-by-query basis. As we materialize data, we know how long we need to keep the data for each query. As we perform commands to create and modify data in a service, we also know how long we need to keep that data to support the usage scenarios of the specific service. Therefore, as we insert and update data, we should set a **time to live** (**TTL**) for each record. The following code extends the example we have been using in this chapter for populating a materialized view in a listener function:

```
  toUpdateRequest: (uow) => ({
     Key: {
       pk: uow.event.thing.id,
       sk: 'Thing',
     },
     ...updateExpression({
       name: uow.event.thing.name,
       ttl: uow.event.timestamp + (60*60*24*33) // 33 days
     }),
   }),
```

To support lean tables, we simply need to add a `ttl` field to each record. In this case, the value is calculated based on `uow.event.timestamp`, which

indicates when the business event occurred. When performing commands to create and modify a domain entity, it is best to use data within the entity to calculate the TTL, such as the expected closing date of a mortgage application. The appropriate duration should be determined on a case-by-case basis. In this example, the useful life of the data for the hypothetical service is 33 days.

More and more databases, such as Amazon DynamoDB, are providing TTL features. This should be an important criterion when selecting a database technology for a service. We simply add the `ttl` field and the database handles, removing the data after it expires. Other services, such as Elasticsearch, support rolling indices on an hourly, daily, or weekly basis, and we can schedule functions to delete older indices. Otherwise, your scheduled function can query and delete individual records based on the `ttl` field.

Removing older data makes tables lean by keeping the number of records relatively consistent. It also helps reduce the impact of changes in upstream services. As we just covered in the *Projections* section, if an upstream change affects one of the projected fields, we need to make a change for all new records. However, we may not need to convert existing records if the TTL is short enough. The queries can continue to support the old and new format until all the old data has expired, and then it is safe to remove the legacy code. In *Chapter 11*, *Choreographing Deployment and Delivery*, we will see this in detail when we cover zero-downtime deployments.

Setting a TTL for records plays an important role in fighting data gravity by controlling the volume of data. This, in turn, makes it easier for systems to evolve. However, we must take care not to remove old data too aggressively as this can have an impact on the idempotence and order tolerance of the system. This is what we will cover next.

Implementing idempotence and order tolerance

In *Chapter 4, Trusting Facts and Eventual Consistency*, we learned that *exactly-once* delivery of messages is unrealistic. For example, a client request may time out due to network unreliability and have no choice but to resubmit the request because it cannot be certain that the service successfully processed the request; a stream processor may fail in the middle of a batch and retry the entire batch even though part of the batch was successfully processed; or we may replay events from the event lake to repair a service that may have dropped a subset of those events. To account for the reality of *at-least-once* delivery, we must design our systems to be idempotent. In other words, no matter how many times we receive and process an event or request, it must only update the system once.

We also learned that delivering messages in order can be problematic. A stream will certainly deliver events in the order that it received them, but it may not receive them in the correct order. For example, a poorly behaving service may produce events in the wrong order; high traffic volumes could cause a service to delay sending events, while related services are still producing their events in real time; or a poison event could be set aside and resubmitted later after we resolve the underlying problem. To account for this reality, we must design our systems to be order tolerant.

In other words, receiving events in order is good for throughput performance, but we must be ready to receive events out of order and literally sort things out.

To achieve this level of resilience, first and foremost we need deterministic identifiers. From here we can employ techniques such as inverse optimistic

locking and immutable event triggers.

Deterministic identifiers

It is a very common practice to delegate the generation of identifiers to the server side, such as by using an auto-incrementing sequence number generated by the database. This is a convenient approach, but it is not idempotent. For example, if a client submits an **HTTP POST** request multiple times for the same logically unique domain entity payload, then the service will create duplicate records for the same entity with different identifiers. The solution to this problem is to have the client generate the identifier and submit an **HTTP PUT** request instead. This will ensure that the request is idempotent and will only create and update one record no matter how many times the client resubmits the request.

There are a variety of alternative techniques for generating an identifier on the client side. One seemingly simple option is to generate a **Universally Unique Identifier** (**UUID**). For domain entities, a V4 UUID is most appropriate because they are generated with random numbers that help avoid hot partitions. For domain events, a V1 UUID is most appropriate because they are generated based on time, which allows them to be naturally sorted.

However, there is a limitation to the UUID approach. It falls apart if the client forgets the UUID that it just generated. For example, if a stream processor is publishing events and fails in the middle of a batch and retries, then it will publish events again with a new UUID. A similar situation happens in a SPA if the client does not remember the identifiers that it generates.

To solve this problem in a SPA client, we can leverage session consistency, as we discussed in *Chapter 3, Taming the Presentation Tier*. With this

approach, the client holds everything in session storage for the duration of the user session. We can go a step further with an offline-first approach and make the data durable in local storage, which we also covered in *Chapter 3*, *Taming the Presentation Tier*.

Another option is to create an identifier by concatenating together a unique combination of fields from the contents of the domain entity. A major advantage of this approach is that the identifier has business meaning. This can be helpful when trying to debug a problem in the business logic. However, it is important to treat these identifiers as opaque values in the code. Otherwise, we may introduce bugs if we change the generation of the identifier.

There is a scenario when it does make sense to use an identifier that the database generates. As we will discuss shortly in the *Leveraging change data capture* section, we can create stream processors that process the stream of change events that a datastore emits as we insert, update, and delete records. These events will have identifiers that the database generated. It is safe to reuse these for the identifiers of domain events because these values will not change, no matter how many times a stream processor retries. However, if you are producing more than one domain event from each database event, then you will need to add a deterministic suffix to these identifiers for each different event type to make them unique.

And finally, another approach is to generate the hash of the entire domain entity or domain event. This is a simple approach, but there is a chance that it will generate collisions. This will depend on the complexity and variability of the domain entity, so consider this option carefully.

Inverse optimistic locking

From the frontend perspective, we use a traditional optimistic locking technique to prevent multiple users from updating the same record concurrently. We store a timestamp or counter in an **oplock** field and we only allow a user to update the record if the oplock value has not changed since the user retrieved the data. If another user has updated the record and thus the **oplock** value, then we throw an exception when the current user attempts to update the record and force the user to retrieve the data again before proceeding with the update. This ensures that we perform updates sequentially, and it requires human interaction to resolve any potential conflicts. We can use this traditional technique in BFF services command functions when users modify data.

Conversely, we use the *inverse optimistic locking* technique in downstream asynchronous stream processors to provide both idempotence and order tolerance. It ensures that older or duplicate events do not overwrite the most recent data. Instead of forcing a transaction to retry when an oplock fails, we simply do the inverse; we drop incoming events if the timestamp is older or equal to the current oplock value in the datastore.

The following DynamoDB example shows the typical structure of a mapping step for a stream processing pipeline that creates a materialized view. We are using `uow.event.timestamp` as the new `oplock` value. `ConditionExpression` compares the old `oplock` value against the new value. If the new value is greater than the old value, then we allow the update to proceed. The `attribute_not_exists` clause handles the case where the record is new and there is no old `oplock` to compare against:

```
export const toUpdateRequest = (uow) => ({
  Key: {
    pk: uow.event.thing.id,
    sk: 'thing',
```

```
    },
    ExpressionAttributeNames: {
      '#name': 'name',
      '#oplock': 'timestamp',
    },
    ExpressionAttributeValues: {
      ':name': 'Thing One',
      ':timestamp': uow.event.timestamp,
    },
    UpdateExpression: 'SET #name = :name,
                           #oplock = :timestamp',
    ConditionExpression: 'attribute_not_exists(#oplock)
        OR :timestamp > #oplock',
  });
```

The first thing to notice in this example that is different with an inverse `oplock` is that the comparison is greater than instead of equals. If the value is equal to the previous value, then this indicates that we have already processed this event. If the value is less than the previous value, then this indicates that we are processing an older event out of order. When this value is greater than the previous value, we know we have a newer event, so we can proceed with the update.

The last piece of the puzzle is handling the response value returned from the call to update the database. The AWS SDK will throw an exception when `error.code` is returned. We don't want to ignore all errors, so we catch the exception and check for the `ConditionalCheckFailedException` code. We re-throw all other exceptions for fault handling, as we covered in *Chapter 4*, *Trusting Facts and Eventual Consistency*. When we do ignore an exception, it is good practice to log a metric so that we can track how frequently this happens and assess whether or not there is a root cause that we should address:

```
db.update(params).promise()
  .catch((err) => {
    if (err.code !== 'ConditionalCheckFailedException') {
      throw err;
    }
  });
```

When using the inverse optimistic locking technique, it is important to ensure that we are not dropping important data when we ignore older events. This could happen if we are using the same `oplock` field for different event types. For example, if the older event is of the same event type, then we know it carries the same data fields as the newer event, so we can drop it safely. However, if the older event is of a different type, then it probably has different fields that we want to capture on the same record, so we do not want to drop this older event.

This is a very typical scenario when implementing the CQRS pattern. We are essentially materializing a join record from multiple types of events. The different event types carry the same value that we are using for the identifier of the join record, plus additional fields per event type that we want to join together. If we were to use the same `oplock` field for all these event types, then we would have to receive them in the correct order.

The solution for joining multiple event types into the same join record is to use a different `oplock` field for each event type. We can refer to this as an **Inverse Oplock Join**. This allows us to incrementally build up these join records one event at a time in any order. Whichever event arrives first will create the record skeleton with the information it carries. Each additional event will flesh out the record with the additional information they carry. Within each event type, we can safely ignore the older or duplicate events.

As with most things, the inverse optimistic locking technique is not right for all scenarios. Let’s take a look at immutable event triggers to see where they fit into the equation.

Immutable event triggers

One of the advantages of the inverse optimistic locking technique is that it doesn’t require the stream processor to retrieve any data to perform a join. This improves the throughput of the stream processor and reduces the possibility of errors. However, it is not always possible to calculate the correct answer without performing a query to retrieve additional information. To get the best of both worlds, we can leverage event sourcing by adding a micro event store to the equation and splitting the logic into two separate stream processors.

The **first stream processor** collects events of interest from the event stream and stores them in a micro event store. This is an atomic step that optimizes the throughput, minimizes the likelihood of errors, and, most importantly, provides idempotence. Event sourcing facilitates idempotence because events are immutable. We can put the same event with the same deterministic identifier into the micro event store over and over again, yet only produce one record.

The following DynamoDB example shows the `put` operation for storing an event in a micro event store. This `put` statement follows the single table design, which we will cover shortly in the *Modeling data for operational performance* section. We use `uow.event.id` as the identifier (`pk`) for the record and we store the immutable event in the `event` field. We will address the `data` field shortly, but the `ttl` field deserves special attention:

```
db.put({
  TableName: options.eventTableName,
  Item: {
    pk: uow.event.id,
    sk: 'event',
    data: uow.event.thing.id,
    event: uow.event,
    ttl: uow.event.timestamp + (60*60*24*33) // 33 days
  }
});
```

In the *Keeping data lean* section, we addressed the importance of not letting our operational datastores grow unbounded. Setting a TTL for each record is the primary technique for controlling the size of a table. However, we must take care to ensure that `ttl` is not too short because this will impact the length of time that the micro event store can enforce idempotence. In this example, `ttl` is set to 33 days. If we receive an event again after this time, then we will process it again. Whether or not it is realistic that this will happen or whether it matters if it does happen is domain-specific, so consider this option carefully. As always, the idempotence and order tolerance of downstream logic must account for this should it happen.

The second stream processor handles the event after we insert it into the micro event store. As we will discuss shortly in the *Leveraging change data capture* section, we can create stream processors that process the stream of events a datastore emits as we insert, update, and delete records. Thanks to the immutability of events, we can be certain that the datastore will only emit the event to the stream once when we first create the record. All additional `put` operations will not change the record and will not emit a datastore event, which provides the processing logic with idempotence.

The second stream processor also provides order tolerance by retrieving related events from the micro event store and reacting based on the events

that have been received thus far. In this example, the `data` field is set to the domain entity's identifier, `uow.event.thing.id`. This allows the stream processor to retrieve all the related events for that domain entity. These events could have arrived in any order, but here the stream processor can sort the events and act accordingly.

For instance, if an expected event is missing, then the processor can do nothing and reevaluates when the next event arrives. Once the necessary information is available, the processor can perform the necessary calculations and save the results to a materialized view.

In the *Calculating event sourcing snapshots* section of *Chapter 8, Reacting to Events with More Events*, you can find a detailed example that uses this technique and **ACID 2.0** properties to perform order-tolerant calculations. Now that we understand the importance of idempotence and order tolerance to the accuracy of our data, let's look at how we can model the data for optimal performance.

Modeling data for operational performance

So far, we have focused on breaking up the monolith and materializing data downstream. Now, we will zero in on modeling data to support the access patterns needed during the operational phases of the data life cycle. This is where various actors (that is, users) create and use data via BFF services. These early phases of the data life cycle need particular attention because they have seen significant changes and improvements.

First, we will look at an approach to logical data modeling that is not coupled to the polyglot persistence choices we make at the individual service

level. Then we will look at horizontal scalability and sharding so that we can understand the implications that this physical model has on our operational data modeling. Then, we will go through several examples of applying the single table design technique to the services in our hypothetical food delivery system.

Nodes, edges, and aggregates

For our operational datastores, we will use serverless, NoSQL databases, such as Amazon DynamoDB. How we model data for these datastores is a little different. To some extent, it is a bit of a paradigm shift. So, it is a good idea to break away from traditional modeling notations, such as relational data modeling, to help us make the shift and avoid falling back into old practices that may produce suboptimal results. We also need a notation that is reasonably neutral with regard to the type of technology that we ultimately use. But we also want a notation that is familiar.

The major distinction between the new and old approaches is that we are more focused on rows than we are on tables. More specifically, these datastores are schemaless, so we are focused on the domain entity types that we store in each row. The types stored in each row will fall into one of two categories: nodes and edges. A **node** represents a domain entity with a unique identifier and an **edge** represents an association between nodes. With this in mind, it makes sense to use a graph-style notation.

In *Chapter 2*, *Defining Boundaries and Letting Go*, we used the event storming technique to help us discover the event types that our system will emit, the relative order of those events, and the actors who produce them. During this process, we also try to identify the data that is in play when we

produce the events, but at that point, we only focus on the domain aggregates.

An **aggregate** represents the root of a set of related domain objects. They provide perspective in the context of the larger data model. They define different starting points for navigating through the nodes and edges of the model. Ultimately, they help identify the access patterns that will be all-important to crafting high-performance, operational data models.

The aggregates also form the starting point for crafting the full logical data model. *Figure 5.6* depicts the logical model for our hypothetical food delivery system. It fleshes out the various domain entities and the associations between them. It is important to note that this is a logical model, and we will see how we can map it to the chosen datastores in a moment:

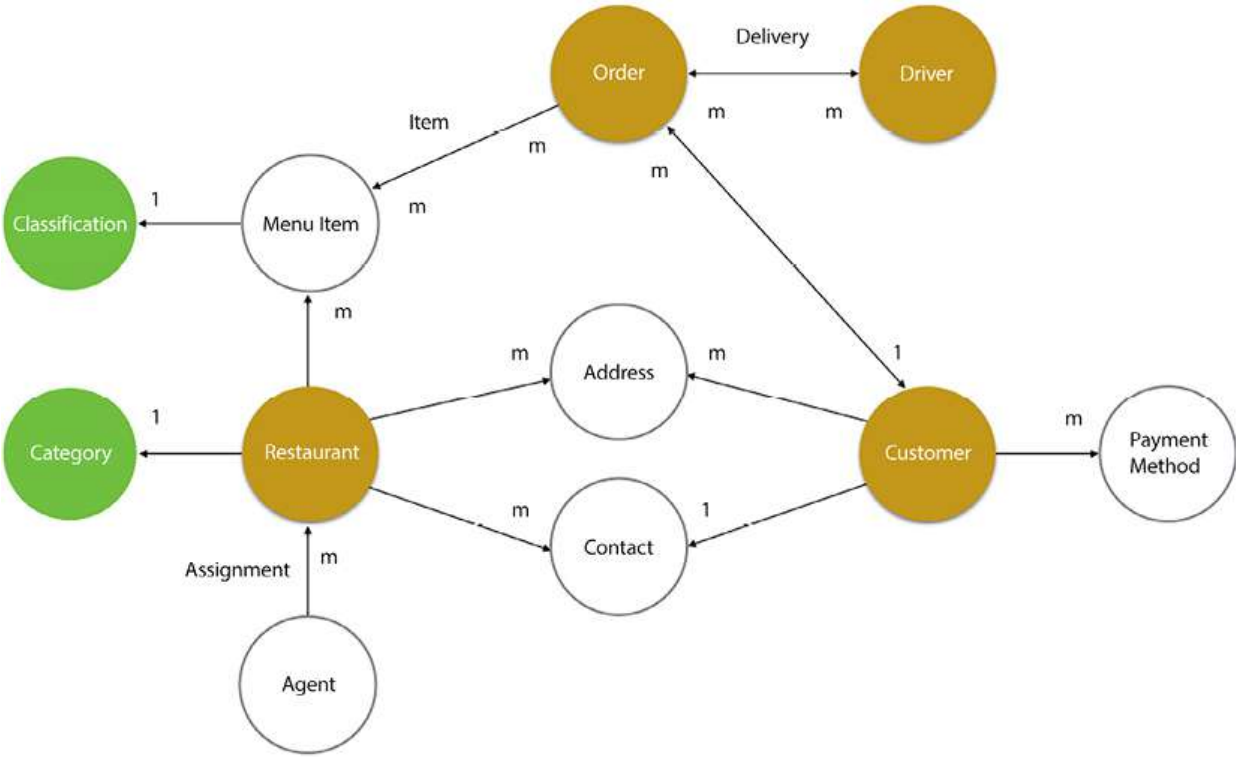


Figure 5.6: Logical data model

The first thing to note in the logical model is that we highlight the aggregates in tan to show their importance and to indicate the various starting points for navigating through the model. We use the same colors that we use in event storming to help form the connection between the models. The aggregates will ultimately become node rows in the physical model.

The other circles represent domain entities that may become node rows in the physical model, or we may embed them as immutable data in other rows depending on how we use them in a specific service.

There are several named associations in the model: **Assignment**, **Item**, and **Delivery**. These will most likely become edge rows in the physical model. The other associations will likely manifest themselves in the primary keys used in the physical model or as embedded objects. It is important to note that the cardinalities in the logical model are for the model as a whole, but in the context of a specific access scenario, they may be different. For example, a customer has many (m) addresses, but in the context of an order, we only use one. It is often worth creating additional models per aggregate to help identify these differences without cluttering the higher-level model.

Before we look at examples of mapping the logical model to a specific datastore, let’s cover sharding and partition keys. This will help explain the choices we make at the physical level.

Sharding and partition keys

The fundamental job of a database is to write data to disk. The disk drive imposes the ultimate limit on how fast we can write and read data and how many concurrent requests we can support. **Sharding** is the process of horizontally partitioning a database across many disk drives. This allows a database to support more concurrent requests. The database keeps track of

which data is on which disk and evenly distributes the data across the disks to help balance the load.

On the application side, we need to properly select and use partition keys to effectively use a sharded datastore. The database uses a **partition key** (**pk**) to match a record to a partition. It stores all the records for a specific pk on the same disk. This means we can optimize performance by modeling our operational data so that we do not need to join data across multiple disks.

We must also take care to avoid hot pks that are not well balanced across partitions. For example, if we use the current date as a pk, then all writes on a given day will go to the same disk and the system will achieve no additional concurrency from sharding. A V4 UUID is a good choice for pks because they are generated with random numbers that result in a good distribution across partitions. We covered additional considerations for pks in the *Deterministic identifiers* section.

Sharding is the driving force behind the popularity of modern NoSQL databases. It also significantly increases the complexity, effort, and cost of running these databases. For example, if you need to replicate data across three cloud availability zones and you need 10 shards, then you have 30 disk drives to manage. If you have 10 services with similar configurations, then that number jumps to 300 disk drives.

You should be running across two regions, so the number of disk drives doubles to 600. Just the effort alone to manage these disks makes the case for leveraging fully managed serverless datastores, such as DynamoDB.

Now, let’s look at some examples that use the single table design technique to fully leverage the power of sharded datastores by grouping related data within the same partition.

Single table design examples

Let’s take the logical data model for our hypothetical food delivery system and see how we can apply the single table design technique to implement service-specific models that optimize performance for the specific actor’s access patterns. These examples are tailored for DynamoDB, but this technique is applicable to many NoSQL databases.

First and foremost, the Single Table Design technique leverages the schemaless nature of these databases. This means that each row in the table can represent a different type of domain entity. Following our graph-oriented logical data model, we can think of each row as either a node or an edge. A **node** represents a domain entity with a unique identifier and has a set of attributes. An **edge** represents an association between nodes and may also contain a set of attributes. We use a discriminator value to help the code determine the domain type of each row. The discriminator may be implemented as a separate field or embedded in the **sort key** (**sk**). The following examples use a separate **discriminator** field for clarity.

It is important to note that while schemaless databases do not use an *explicit* schema, there is an *implicit* schema associated with each row. We manage these implicit schemas within the code. In *Chapter 11*, *Choreographing Deployment and Delivery*, we will see how we do this and why it makes it easier to perform continuous, zero-downtime deployments.

The next and equally important element of the technique is the **pk**. As we have already seen, a pk maps a record to a specific shard (that is, disk). We want to leverage this fact and use the pk field to pre-calculate a join between different records by storing them all on the same shard. This allows us to retrieve all the related data with a single, highly efficient query. It is helpful to think of the pk as the unique identifier for a domain **aggregate** that

represents a collection of related domain objects that we want to work with as a group, such as the data for a restaurant, customer, order, or driver in our logical model. Within a pk, we use an **sk** to uniquely identify each of the related records.

The pk defines our primary access pattern, which is the ability to retrieve all the data for a specific domain aggregate. However, we often need to access the same data from a different direction, across aggregates, such as implementing a basic search query. We use the **data** field for these types of queries and create an index on this field. In DynamoDB, we create a global secondary index. We can think of this as an alternate pk that organizes the data on a different set of partitions so that we can access it with a single query as well.

Let’s look at some single table pattern examples.

Restaurant service

This first example supports agents of the delivery company who manage menus on behalf of restaurants:

discriminator	pk	sk	data	fields		
Assignment	agent1	r1		Mamma Marie's		
Assignment	agent1	r2		29 Diner		
Category	cat1		meta	Italian	icon font	
Category	cat2		meta	Greek	"	
Classifier	class1		meta	Breakfast	"	
Classifier	class2		meta	Dinner	"	
Restaurant	r1		cat1	Mamma Marie's	address {…}	
Restaurant	r2		cat2	29 Diner	"	
MenuItem	r2	r2|mi1	class1	Omelet, $7.99	options {…}	image url
MenuItem	r2	r2|mi2	class2	Cheeseburger, $6.99	"	"

Table 5.1: Restaurant service example data

Here is an explanation of the different nodes and edges:

- An **agent** is a logical node that we do not store in this table because agents already know who they are. When agents log in, they receive their employee ID as a claim in their JWT token, such as **agent1**. A different service will handle the management of assignments and that service will need to store agent nodes for the manager to choose from.
- **Assignment** edges map agents to the restaurants they support. We materialize these records from upstream events. We use the current employee ID (that is, **agent1**) to query the table on the **pk** field to retrieve the data needed to display a list of restaurant assignments for the specific agent. In this case, **agent1** has assignments for restaurants **r1** and **r2**.
- **Category** and **Classifier** nodes contain metadata that we use for grouping restaurants and menu items. The application displays these as drop-down menus in the agent's data entry screens. The application can retrieve all this data with one simple query on the **data** field for the value **meta**.
- **Restaurant** nodes contain information about each restaurant, such as its name, address, and category.
- **MenuItem** nodes contain information about the food a restaurant offers, such as its name, price, options, picture, and classifier.
- An agent can retrieve all the data for a specific restaurant, such as **r2**, with a single query on the **pk** field.
- An agent can also query for similar restaurants and menu items with a query on the **data** field for the value of a category or classifier. We prefix the menu item **sk** with the restaurant identifier to make them unique within this index.

In the *Leveraging change data capture* section, we will see how to produce domain events from this data to support downstream services. In *Chapter 6*, *A Best Friend for the Frontend*, we will see how a downstream menu service could react to these events and provide a responsive and resilient customer experience using datastores such as AWS S3 and Elasticsearch.

Customer service

This example allows customers to maintain the personal information that they need to create food delivery orders. It also supports viewing a customer's order history:

discriminator	pk	sk	data	fields		
Customer	c1			first-name	last-name	
Contact	c1	e1		contact {…}		
Address	c1	a1		address {…}		
Address	c1	a2		"		
Card	c1	p1		card {…}		
Order	o1		c1|2020	order {…}		
Order	o2		c1|2020	"		

Table 5.2: Customer service example data

Here is an explanation of the different nodes and edges:

- When customers sign in, they receive their customer ID as a claim in their JWT token, such as **c1**. We use the customer ID to retrieve all the customer's data with a single query on the **pk** field. However, this won't include order history because we do not need that data at the same time.
- **Customer** nodes allow customers to maintain their summary information, such as their first and last names. We store the customer

ID in the **pk** field.

- **Contact** nodes allow customers to maintain their phone numbers and email addresses. The **sk** field contains the unique identifier and we link it to the customer via the **pk** field.
- **Address** nodes allow customers to maintain the physical locations where they want food delivered. The **sk** field contains the unique identifier and we link it to the customer via the **pk** field.
- **Card** nodes allow customers to maintain their payment information. The **sk** field contains the unique identifier and we link it to the customer via the **pk** field.
- **Order** nodes contain historical information about a customer's orders. These orders are immutable, so we embed the order details within each record. Customers can retrieve their order history by year with a query on the **data** field with the customer ID and year, such as **c1|2020**.

Cart service

This example allows customers to prepare their current order in a cart. It does not retain the order history. We have already seen how we can retain the order history in the customer service example:

discriminator	pk	sk			
Order	c1	o1	2020-10-01	$29.99	
Restaurant	c1	o1|r1	address {…}	contact {…}	
Item	c1	o1|i1	qty	price	options {…}
Item	c1	o1|i2	qty	Price	options {…}
Customer	c1	o1	address {…}	contact {…}	card {…}

Table 5.3: Cart service example data

Here is an explanation of the different nodes and edges:

- When customers sign in, they receive their customer ID as a claim in their JWT token, such as **c1**. We use the customer ID to retrieve the contents of the customer's cart with a single query on the **pk** field.
- **Order** nodes contain summary information, such as the date of the order and the total cost. The **sk** field contains the unique order identifier and we link it to the customer via the **pk** field.
- **Restaurant** nodes contain the address and contact information of the restaurant. We retrieve this information from the menu service as the user browses and we save an immutable copy per order. The **sk** field contains the unique restaurant identifier prefixed with the order identifier and we link it to the customer via the **pk** field.
- **Item** nodes contain the items the customer wants to order from the restaurant's menu. We retrieve menu item metadata from the menu service as the customer selects an item and we store an immutable copy per item along with any customization options the customer has selected. The **sk** field contains the unique item identifier prefixed with the order identifier and we link it to the customer via the **pk** field.
- **Customer** nodes contain the address, contact, and payment information the customer has selected for the delivery. We retrieved this information from the customer service and we save an immutable copy per order. The **sk** field contains the order identifier and we link it to the customer via the **pk** field.

Once the customer submits an order, we can flag these records for deletion with a short TTL. From here, an order tracking service will maintain a materialized view that provides the customer with order status information. I'll leave that to you as an exercise. We have already seen how the customer service retains the order history.

Delivery service

This example provides drivers with the information they need to execute a delivery. Drivers can search for unassigned orders by zone, acquire a delivery assignment, and view the order details:

discriminator	pk	sk	data		
Order	o1			order {…}	status
Delivery	zone1	o1	d1	Mamma Marie's	geo locations
Delivery	zone1	o2	unassigned	29 Diner	"

Table 5.4: Delivery service example data

Here is an explanation of the different nodes and edges:

- **Order** nodes contain an immutable copy of the order so that drivers can retrieve the order details. The **pk** field contains the unique order identifier.
- **Delivery** edges associate orders with delivery zones and drivers. We include the restaurant name to provide additional context information to drivers, along with the geo-locations for navigation to the restaurant and the customer. The **sk** field contains the unique order identifier and we link it to the zone via the **pk** field. We track the assigned driver in the **data** field.
- Drivers can query the table by zone to find unassigned deliveries and acquire them by setting the **data** field to their driver identifier, such as **d1**.
- Drivers, such as **d1**, can query all their current deliveries with a single query on the **data** field.

This delivery BFF service is fairly rudimentary. A **Control Service**, such as those we cover in *Chapter 8, Reacting to Events with More Events*, could

match potential drivers to deliveries, and an **External Service Gateway** (**ESG**) service, such as those we cover in *Chapter 7, Bridging Intersystem Gaps*, could react by sending SMS messages to the drivers. A driver BFF service could track a driver's delivery history, similar to how the customer service tracks the order history.

From these examples, it should be clear how turning the database inside out enables the creation of responsive and resilient services that are much easier to maintain and evolve. So far, we have focused on the downstream side, receiving domain events from upstream services and materializing the needed data for use by a specific service. Let's switch to the upstream side and see how those services can leverage change data capture streams to produce their domain events.

Leveraging change data capture

Let's change gears a little bit. Up to this point in this chapter, we have primarily focused on the downstream consume and query side of the CPCQ flow. Now, we will look into the upstream command and publish side of the CPCQ flow and how it intersects with the system-wide Event Sourcing pattern that we introduced in *Chapter 4, Trusting Facts and Eventual Consistency*.

In the *Turning the database inside out* section, we learned that splitting data out into multiple databases creates the need for a system-wide transaction log. We fulfill this need with the Event Sourcing pattern and the event lake. Each autonomous subsystem plays its role by providing an event hub service to collect events and an event lake service to store them. Individual

autonomous services play their part by publishing events to the hub as their state changes.

Each individual database still has its own transaction log, and we leverage it as a powerful tool. **Change Data Capture** (**CDC**) is a technique that reads a database's transaction log in real time and captures these database-level change events and exposes them to external consumers as a stream so that they can react as the data changes.

CDC plays an important role in turning a database inside out. We can leverage it to implement event sourcing on the upstream command side in a way that simplifies the development experience. Let's see how.

Database-first event sourcing

In the *Dissecting the CQRS pattern* section, we learned that CQRS employs separate read and write data models and that system-wide CQRS takes the approach further by splitting the models between upstream and downstream actors to optimize the experience for each user and allowing them to change independently. To facilitate this approach, the upstream service must emit domain events as the data in its write model changes so that downstream services can materialize the data in their read models.

In *Chapter 4*, *Trusting Facts and Eventual Consistency*, we learned that modern databases do not support distributed transactions because they do not scale cost-effectively. Instead, we create eventually consistent systems by chaining together a series of atomic actions that each update a single datastore. The approach we covered for initiating this chain reaction is the stream-first variation of the Event Sourcing pattern. With this variation, we first emit (that is, write) an event to the event hub, where a stream acts as a temporal datastore so that downstream services can react and materialize

lean data in their own datastores. In other words, the upstream service does not immediately persist its own copy of the data it creates; it only writes to the stream.

Stream-first event sourcing is best suited for fire-and-forget scenarios where the event producer does not need to immediately interact with the outcome of the processing. In *Chapter 7*, *Bridging Intersystem Gaps*, we see that stream-first is often the right approach for ESG services. In *Chapter 8*, *Reacting to Events with More Events*, we see how stream-first is the best approach for Control Services. However, stream-first event sourcing can be problematic for BFF services.

BFF services support end users, and end users typically need to interact with the data they create. This is where the Event Sourcing and CQRS patterns get their bad reputations. The otherwise simple process of writing the current state to the database and later reading it is now spread across two separate models and thus two different development activities. We have already seen how to materialize the read model in the *Dissecting the CQRS pattern* section, so this is not necessarily difficult to do. But it is one more thing to do, so if we can avoid it on the upstream command side, then it is worth considering.

This is where CDC can help. CDC provides us with an opportunity to have the best of both worlds. We can have the streamlined developer experience of writing and reading the current state with a single data model, while also atomically producing domain events so that downstream services can support their own read models. This is the database-first variation of the Event Sourcing pattern:

Figure 5.7: Database-first event sourcing

Figure 5.7 depicts a **BFF** service using the database-first variant of event sourcing. The **command** function performs the single atomic action of writing the outcome of its processing to a **table** that it owns. The CDC feature of the database, such as Amazon DynamoDB Streams, makes the database change event visible to the **trigger** function. The **trigger** function transforms the change event and produces a domain event to the **event hub**.

It is important to recognize that with database-first event sourcing, we are relying on the at-least-once semantics of stream processing to help guarantee that we eventually publish the domain events. As with all streams, a CDC stream is a temporal datastore. In the case of an unexpected error, the **trigger** function will continue to retry until we successfully process the change event or it expires, and in the case of an expected error, the **trigger** function will produce a fault event. We covered this topic in the *Designing for failure* section of *Chapter 4, Trusting Facts and Eventual Consistency*.

Now let’s look at how we implement the stream processing of a **trigger** function. The following example builds on the concept of pipeline patterns, which we covered in *Chapter 4, Trusting Facts and Eventual Consistency*:

```
import { cdc } from 'aws-lambda-stream';
const RULES = [{
  id: 'a1',
  flavor: cdc,
  eventType: /thing-(created|updated|deleted)/,
}, {
  id: 'd1',
  flavor: cdc,
  eventType: 'thing-updated',
  filters: (uow) =>
    uow.event.raw.new.status === 'submitted',
  toEvent: (uow) => ({
    type: 'thing-submitted',
    thing: {
      id: uow.event.raw.new.pk,
      name: uow.event.raw.new.nm,
      description: uow.event.raw.new.descr,
    },
  }),
}];
```

This example emits domain events as we insert, update, and delete data. The typical steps for creating domain events from a CDC stream are as follows:

1. Filter for the desired database change events.
2. Map the data in the database change event to the desired domain event.
3. Publish the domain event to the event hub.

These steps are so typical that it makes sense to wrap them into a reusable pipeline pattern. This example uses a `cdc` pipeline pattern that works with databases such as DynamoDB that support CDC. And of course, you can create your own or implement the pattern in your favorite language and framework.

NOTE

The mapping code in this example is included inline for brevity. These functions will normally be implemented and tested in separate modules.

The most interesting details are in the `toEvent` functions, which implement the mapping step. The first rule, `a1`, uses a regular expression to filter for created, updated, and deleted events for the thing domain entity. This rule does not define a mapping function, so it will publish the database change events as is. Publishing the raw database change events serves two purposes. First, these events act as an audit log of exactly how the data has changed, when it changed, and who changed it.

The `uow.event.raw` element includes the new and old images of the data just as they were recorded by the database's transaction log. In *Chapter 10*, *Securing Autonomous Subsystems in Depth*, we will see how a command can pass the current user from a JWT along to the database layer to track which users are modifying which data.

As we covered in *Chapter 4*, *Trusting Facts and Eventual Consistency*, this raw element is for internal use only and the contents are not guaranteed to be backward-compatible. That said, the second benefit of publishing the raw database change events is to jumpstart the development process. Within a new and evolving autonomous subsystem, we can use these database change events to collaborate between services and help discover the true domain events that we should publish.

The second rule, `d1`, in this example publishes an explicit domain event. It specifically filters for the thing-updated change event and then includes an additional filter for when the `uow.event.raw.new.status` field is set to `submitted`. A `status` field is a typical approach used to track the state of a

domain entity as it progresses through a workflow. It allows users to search for their work based on the status and it allows other users to determine the current status of work. We can also leverage the `status` field in our CDC pipelines, as is the case here, to determine which domain event to publish.

The `toEvent` mapping function sets the event type value to `thing-submitted` and maps the internal database fields to the contents of the thing element. These fields form the contract with downstream services, and we must ensure that any changes are backward-compatible. We will cover the versioning of domain events in *Chapter 11*, *Choreographing Deployment and Delivery*.

We have already seen how CDC plays a part in implementing idempotence and order tolerance. CDC is also an invaluable approach for integrating with legacy systems, as we will see in *Chapter 7*, *Bridging Intersystem Gaps*. We will also see how we can leverage CDC to integrate multiple datastores within a single service in *Chapter 6*, *A Best Friend for the Frontend*. In *Chapter 8*, *Reacting to Events with More Events*, we will see how CDC plays a role in orchestration and complex event processing.

Now, let's look at the important concepts of soft deletes and latching.

Soft deletes

On the surface, deletion seems like a simple command, but it has some very interesting implications. Certainly, deleting data can have security ramifications, so we want to keep an audit trail as usual. A user could delete a record by mistake, so we should be able to undo this destructive action. But there are also some additional nuances with the `delete` command that arise in eventually consistent systems.

Idempotence and order tolerance can be difficult to achieve when we perform a hard delete. For example, if an updated event arrives after a deletion event, then we would recreate the record and leave the datastore in an inconsistent state. A similar situation can happen when replaying events. We could temporarily add a deleted record back and then remove it again. The end result here is eventually consistent, but a query in the meantime could return confusing results.

Another nuance arises when we have slow data that we want to resynchronize downstream. In this case, an upstream service maintains data, such as reference data or master data, which changes very infrequently. Downstream services will cache this data and it may become stale, such as when they have defects. For example, they may drop a deletion event and continue to present the deleted data. For reasons like this, the upstream services that own slow data will produce `resync` events periodically or on demand. These `resync` events need to include information about data that was recently deleted. But how can we do this if the data is deleted? We can certainly replay these events from the event lake. However, they will be spread out through the lake and be time consuming to process because the data is slow, and the events could have happened a long time ago.

The solution to these problems is to perform soft deletes instead of hard deletes. A hard delete physically removes the data, whereas a soft delete just flags the data as deleted. The following DynamoDB example shows how a `delete` command would simply set a `deleted` flag on the existing record. Any corresponding queries would filter out all records that we flagged as deleted:

```
db.update({
  Key: {
    pk: thing.id,
```

```
      sk: 'Thing',
    },
    ExpressionAttributeNames: {
      '#deleted': 'deleted',
    },
    ExpressionAttributeValues: {
     ':deleted': true,
    },
    UpdateExpression: 'SET #deleted = :deleted',
  });
```

On the CDC stream, the change event will appear as an update. However, the `trigger` function will see that the `deleted` flag is set to `true` and publish a `thing-deleted` event instead. This trigger logic is implemented as a feature in the cdc pipeline pattern.

It is important to keep data lean, so we also need to decide how long to keep records that we flag as deleted. For slow data with limited records, it may make sense to keep these records indefinitely. Otherwise, we can set a TTL as we discussed in the *Keeping data lean* section. When the TTL expires, a hard delete will be performed. This time, the CDC stream will emit a deleted event. The trigger function can ignore it or publish another `thing-deleted` event for good measure. Publishing it again would give downstream services another opportunity to delete their record or idempotently ignore the duplicate event.

When a service emits resync events, it can include events for all records that we flagged as deleted. We will revisit slow data in *Chapter 7, Bridging Intersystem Gaps*. Now, let's look at latching.

Latching

To enable continuous change, an evolutionary architecture must support running multiple versions of functionality concurrently. We will see this need in Chapter 7, *Bridging Intersystem Gaps*, when we are incrementally modernizing a legacy system. But even the new system will evolve as new requirements emerge and as teams find better ways to satisfy existing requirements. In many cases, a service can simply evolve in place, and we will cover this in *Chapter 11*, *Choreographing Deployment and Delivery*.

However, much like a legacy migration, there are times when an evolution is more of a revolution. In these cases, to mitigate risks, it is best to create a new version of a service that runs concurrently with the old version. This gives beta users an opportunity to weigh in on the new version of the feature and provide valuable feedback.

To accomplish this, we need to synchronize the data between the two versions of the service. This will allow several scenarios to potentially unfold:

1. If beta users think the new version has missed the mark, then they can switch back to the old version and pick up where they left off.
2. If the functionality needs to be migrated incrementally, then users can seamlessly perform some work in the new version and some in the old version.
3. Finally, when we release the new version to all users, the data will already be converted, and we can remove the old version.

When implementing a bi-directional synchronization like this, we must take care not to create an infinite loop. For example, the trigger function of Service A produces an event and it is consumed by the listener function of Service B. The trigger function of Service B then produces the same logical event and it is consumed by the listener function of Service A. The trigger

function of Service A produces the event again and the loop continues infinitely.

Latching, much like locking a door, is the solution to this problem. The `trigger` function will not emit an event when the latch is closed. The `listener` function will close the latch when it materializes data, and a `command` function will open the latch when a user of the service performs an action. This prevents synchronization updates for looping but allows user-initiated state changes to be emitted.

The following code extends the example we have been using in this chapter for populating a materialized view in a `listener` function:

```
  toUpdateRequest: (uow) => ({
     Key: {
       pk: uow.event.thing.id,
       sk: 'Thing',
     },
     ...updateExpression({
       ...uow.event.thing,
       latched: true,
     }),
   }),
```

The `listener` function closes the latch by setting the `latched` field to `true`. This field tells the `trigger` function whether or not to emit a domain event. Since the value is set to `true`, the trigger function will not emit a domain event and we avoid an infinite loop. This trigger logic is implemented as a feature in the `cdc` pipeline pattern.

When the user of this BFF service performs a command that updates the data, the command opens the latch by setting the `latched` field to `false` or

removing the field. Since the value is `false` or the field is absent, the `trigger` function will emit the domain event.

Back on the `listener` function side, a service can safely ignore the events that its trigger function just emitted. To accomplish this, we filter on the `source` tag value. Every event should contain a `source` tag that we set to the name of the service that emitted it. The `listener` function simply ignores any events with the `source` value set to its own service name.

This last point brings up an interesting difference between traditional and serverless systems. In a traditional system, we might not be concerned about the `listener` function updating the record again with the same values. We might say there's no harm, so there's no foul. However, there is a cost because the additional update consumes capacity.

In a traditional system, this cost is paid for by the end users in the form of decreased performance. In a serverless system, the cost increases the monthly invoice and gives us a stronger incentive to do the right thing by avoiding unnecessary updates.

Leveraging CDC to implement database-first event sourcing is a very powerful tool that greatly simplifies the developer experience by separating the different concerns, while also producing a more responsive and resilient user experience. We will use this pattern throughout the book.

Summary

In this chapter, you learned why data gravity impedes innovation and how to decompose monolithic databases throughout the phases of the data life cycle. You learned how the CQRS pattern turns the database inside out and ultimately turns the cloud into the database by creating a system-wide

transaction log and moving derived data downstream to where it is used. You also learned how to increase team confidence by using materialized views to create inbound bulkheads.

We dug into the details and you learned how to create stream processors that are idempotent and order tolerant, and you saw how to keep data lean. You also learned about using the Single Table Design modeling technique to optimize data for performance and how to create stream processors that leverage change data capture to implement the database-first variant of the event sourcing pattern.

In the next chapter, we will cover the BFF pattern and see how it unencumbers teams by removing competing demands so they can focus on the end user's needs. We will also look at variations of the BFF pattern throughout the data life cycle.

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.

**Limited Offer*

6

A Best Friend for the Frontend

In *Chapter 4, Trusting Facts and Eventual Consistency*, we covered the **Event Sourcing** pattern and learned how it creates an outbound bulkhead that protects upstream services from downstream outages. In *Chapter 5, Turning the Cloud into the Database*, we covered the **Command Query Responsibility Segregation** (**CQRS**) pattern and learned how it creates an inbound bulkhead that protects downstream services from upstream outages. Together, these bulkheads fortify the boundaries of autonomous services and help control the blast radius when things go wrong.

Now we turn our attention to the **boundary service** patterns that work at the system's periphery. These autonomous services cater to the kinds of changes driven by external actors, such as end users and external systems. In this chapter, we will focus on supporting end users. You will learn how to create a backend for frontend services that cater to the kinds of changes driven by the human actors who use the system. These services separate the requirements of different actors so that they do not compete and impede innovation.

In this chapter, we're going to cover the following main topics:

- Focusing on user activities
- Dissecting the Backend for Frontend pattern
- Dissecting function-level nano architecture
- Choosing between REST and GraphQL
- Implementing different kinds of BFF services

Focusing on user activities

In *Chapter 1*, *Architecting for Innovation*, we saw that the role of architecture is to enable change so that teams can continuously experiment and uncover the best solutions for their users. We enable continuous change by defining fortified boundaries around things that change together so that we can control the scope and impact of any given change. The key to defining these boundaries is to understand the driving force behind change.

In *Chapter 2*, *Defining Boundaries and Letting Go*, we found that people (that is, actors) are the driving force behind change. We identified three autonomous service patterns that each cater to a different type of actor. In this chapter, we dig into the details of the **Backend for Frontend** (**BFF**) service pattern. The BFF service pattern works at the boundary of the system to support end users and the activities they perform.

Before going further, it is important to put the BFF service pattern into context, because it is easy to get bogged down in all the interrelated details when we build a complex system. There are many features, different users, integrations with external systems, and overarching business processes. We need to break the larger problem down into smaller problems so that we can control the complexity and manage change.

Our three autonomous service patterns control the complexity and the scope of change. We cover integrations and the External Service Gateway pattern in *Chapter 7*, *Bridging Intersystem Gaps*, and we cover business processes and the Control Service pattern in *Chapter 8*, *Reacting to Events with More Events*.

The purpose of the BFF service pattern is to focus on the requirements of end user activities and let all other concerns fade into the background.

Concerns about integrations should not encumber a BFF service. Each user activity is a step in an overarching business process, but only the inputs and outputs of the step are of concern to a BFF service. This level of focus is important because it minimizes coupling and increases flexibility.

Let's look at how the BFF service pattern facilitates change by zeroing in on user activities, self-sufficient teams, and autonomy.

A BFF service is responsible for a single user activity

The BFF service pattern facilitates change because each BFF service has a deliberate focus on a specific user activity and eliminates distractions and competing demands.

Following on from the **Single-Responsibility Principle** (**SRP**), a BFF service is responsible to one and only one human actor who interacts with the system's user interface. However, a single human actor may interact with a system in multiple ways to perform different activities. In this case, having one BFF service that supports multiple user activities could result in competing demands unless the activities are closely related.

In *Chapter 3*, *Taming the Presentation Tier*, we covered multiple approaches for breaking up a monolithic presentation tier, such as dynamically assembling many micro-apps in a micro frontend architecture or creating dedicated mobile applications for specific activities. In *Chapter 5*, *Turning the Cloud into the Database*, we discussed how data requirements change as data moves through its life cycle and interacts with different actors.

Taking all these considerations together, it is best to create a separate BFF service for each discrete user activity. This generally equates to creating a

BFF service for each micro-app or mobile application, hence the term Backend For Frontend. For our food delivery example, we might include the following BFF services:

- A service used by restaurants to create menus, which is different from the service used by customers to browse menus
- A service used by customers to create orders, which is different from the service used by drivers to deliver orders and different still from the service used by customers to track the status of an order
- A service used by customers to view their order history, another used by drivers to view their delivery history, and yet another used by restaurants to view their order statistics
- Separate services for customers, drivers, and restaurants to manage their accounts

Many of these examples seem very similar and their implementations may even look identical at first, but avoid the temptation to merge them. They support different actors and, therefore, different user activities. They belong to different bounded contexts. Before long, they will begin to diverge as the different actors drive different changes. Fortunately, as we covered in *Chapter 2, Defining Boundaries and Letting Go*, the services for the different actors will likely live in different autonomous subsystems, which will help ensure that the services remain separate and change independently.

Sometimes it is necessary to deliver the same user experience with multiple frontend implementations, such as a web application, a native iPhone application, and a native Android application. In this case, we need to decide between whether to support all the implementations with a single BFF service or one per implementation.

If the functionality of all the implementations is identical, then a single BFF service is probably the right choice. If there are significant variations, then multiple BFF services may be better. If separate teams own the implementations, then that alone is a reason to have different BFF services, as we will cover next.

A BFF service is owned by the frontend team

The BFF service pattern facilitates change because the same team owns and implements the frontend and the backend.

This is a critical distinction. Traditionally, we have structured teams for the layers of the architecture with frontend teams, backend teams, and database teams. However, this structure requires significant inter-team communication and coordination, which results in long lead times that impede innovation. Instead, self-sufficient, full-stack, autonomous teams own and focus on the frontend and the backend for one or more related user activities.

This enables a team to follow a UI-first approach. The team rapidly prototypes the user interfaces with no backend, to elicit fast feedback from the end user. In turn, the UI drives the requirements for the BFF service. This approach easily identifies the necessary queries and commands and eliminates the waste that arises when a separate backend team implements features that it assumes a frontend will need.

When a team is ready to implement a BFF service, they will typically leverage the same language used to implement the frontend, such as JavaScript. This allows the same developers to use the same tools to work on

the frontend and the backend. This increases productivity because it reduces context switching as developers move between related frontend and backend tasks. In addition, the BFF pattern can reduce the cost of fielding a frontend team by controlling the number of necessary skill sets.

A BFF service is decoupled, autonomous, and resilient

The BFF service pattern facilitates change because it eliminates inter-service dependencies that create runtime coupling and reduce resilience.

This pattern is relatively new and has taken on various shapes and sizes as it has evolved. One common approach is to implement a BFF as a facade service that wraps and aggregates the calls to other services. But this approach is too brittle. It does eliminate the *design-time coupling* between the frontend and the various backend services, but it does not eliminate the *runtime coupling*. If any one of these other services has an outage, then the frontend user will also experience degraded functionality.

Furthermore, inter-service coupling increases inter-team communication and coordination. A miscommunication between teams can lead to backward-incompatible changes that cause outages.

Instead, we implement our BFFs as autonomous services that eliminate synchronous inter-service dependencies. These services do not wrap other services. They own all the resources they need to continue operating when related services are down. They leverage the CQRS pattern to cache data from upstream services, as we covered in *Chapter 5, Turning the Cloud into the Database,* and they leverage the Event Sourcing pattern to produce events so that downstream services can react, as we covered in *Chapter 4,*

Trusting Facts and Eventual Consistency. Together, these lower-level patterns provide the bulkheads that protect BFF services, so they can deliver the responsiveness and resilience demanded by frontend users.

Let’s look at the anatomy of a BFF service before we dive into some applications of the pattern.

Dissecting the Backend for Frontend pattern

In *Chapter 2*, *Defining Boundaries and Letting Go*, we introduced the concept of micro (that is, service-level) hexagonal architecture. This architecture level describes how we bring multiple functions and resources together to implement a given autonomous service. At the macro level, an autonomous subsystem is composed of many of these autonomous services, so it is important that they all have a similar feel and structure. This will allow developers to easily work on different services. To help achieve this, it is good practice to seed a new service with a skeleton from a standard template, stand it up, and then add the specific functionality. *Figure 6.1* depicts the resources that make up the typical BFF service.

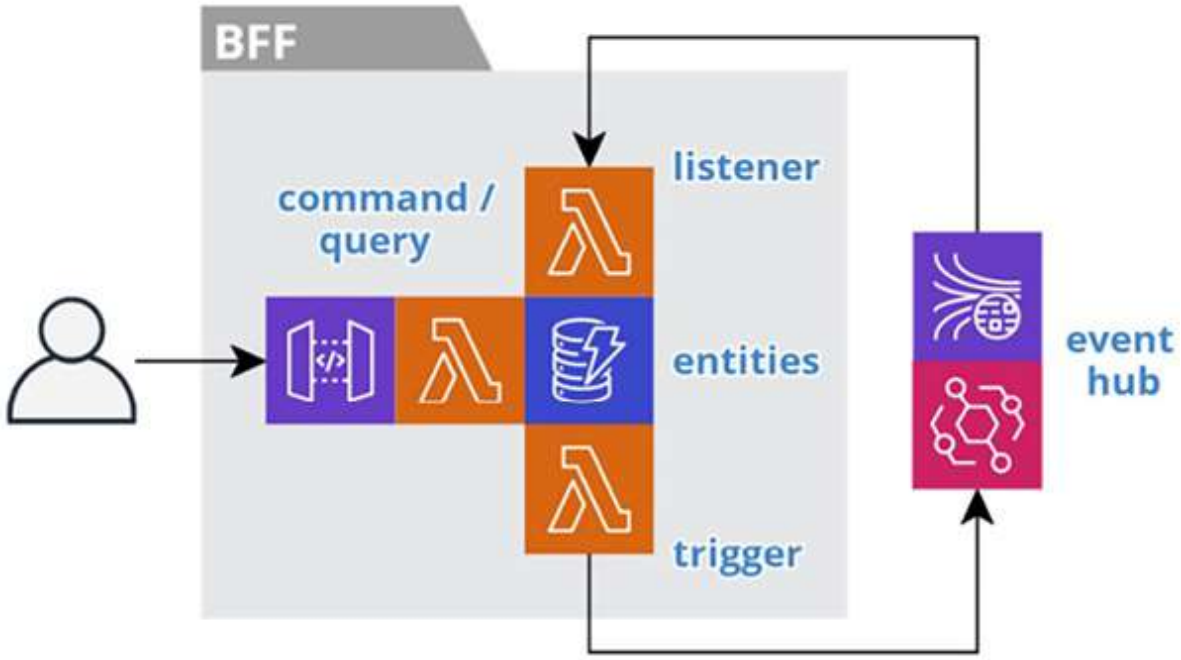


Figure 6.1: BFF pattern

The first thing to notice is that the typical BFF service has a *trilateral API*:

1. It has a synchronous `command` and `query` interface that supports its frontend.
2. It has an asynchronous `listener` interface that consumes domain events from upstream services via the `event hub`.
3. It has an asynchronous `trigger` interface that produces domain events to the `event hub` for consumption by downstream services.

You can find a BFF service template here: https://github.com/jgilbert01/templates/tree/master/template-bff-service.

Let’s dig into the different resources utilized by a BFF service.

Datastore

Each BFF service maintains its own highly available, fully managed datastore, such as AWS DynamoDB or S3. This is how they achieve their autonomy. The datastore caches entities from upstream domain events, so the data is readily available to end users. Users make changes to the entities in the datastore, and the service produces domain events in the background to share the state changes.

It is important to understand that a BFF service stores data in a format that is most suitable for consumption by the frontend. This will produce a user experience that is more responsive and simplifies the frontend code by eliminating the need to continuously transform the data.

The following `serverless.yml` fragment from the BFF service template creates a DynamoDB table that supports the *single table design* best practice:

```
 Resources:
EntitiesTable:
Type: AWS::DynamoDB::GlobalTable
Properties:
TableName: ${self:service}-${opt:stage}-entities
AttributeDefinitions:
...
KeySchema:
- AttributeName: pk
KeyType: HASH
- AttributeName: sk
KeyType: RANGE
GlobalSecondaryIndexes:
- IndexName: gsi1
...
BillingMode: PAY_PER_REQUEST
StreamSpecification:
StreamViewType: NEW_AND_OLD_IMAGES
TimeToLiveSpecification:
```

```
  AttributeName: ttl
  Enabled: true
```

This standard table definition consists of a generic `KeySchema`, which supports all entity types, and one or more global secondary indices. The **Change Data Capture** (**CDC**) streaming and **Time To Live** (**TTL**) features are enabled, along with on-demand billing.

We covered these database topics in detail in *Chapter 5, Turning the Cloud into the Database*.

API gateway

The frontend must have access to the BFF service to perform queries and execute commands. The API gateway provides this capability. It creates a secure and highly available perimeter around the functionality to support the frontend. We will cover this in detail in *Chapter 9, Running in Multiple Regions*, and *Chapter 10, Securing Autonomous Subsystems in Depth*.

It is important to recognize that this interface supports a specific frontend and only that frontend. Other systems should not call this interface, as they are different actors. The External Service Gateway pattern fills this purpose, and we will cover it in *Chapter 7, Bridging Intersystem Gaps*.

The following `serverless.yml` fragment, from the BFF service template, creates an API gateway and routes it to the functions that implement the queries and commands:

```
functions:
rest:
handler: src/rest/index.handle
events:
- http:
```

```
  path: {proxy+}
            method: any
```

This example routes all HTTP methods and paths to a single function. We will cover this function next.

Query and command functions

The all-important job of a BFF service is to provide a human actor with the ability to execute a specific set of queries and/or commands. They retrieve requested data from the datastore, and they record the results of the user's activities in the datastore. We implement this functionality in one or more functions, using the cloud provider's **Function-as-a-Service** (**FaaS**) feature, such as AWS Lambda.

A common question regarding FaaS is whether to use many fine-grained functions or fewer coarse-grained functions. Our user activity-focused BFF services are already fairly fine-grained, so I prefer to start with a single function that executes all the BFF's queries and commands. For a GraphQL-based service, this is a natural fit for the popular Apollo framework. For a REST-based service, I may eventually split out a separate function for queries and one for commands if the traffic volumes are different enough. However, these variations may represent different user activities and warrant separate BFF services instead. As the number of queries and commands grows, this may also indicate that the scope of the BFF service has grown too broad.

We will dig deeper into these functions in the *Dissecting function-level nano architecture* section and the *Choosing between REST and GraphQL* section.

Listener function

The `listener` function plays a crucial role in an autonomous service of implementing the CQRS pattern. It is a stream processor that consumes domain events from the event hub and materializes entities in the datastore to support the queries. BFF services store data in a format that is most suitable for consumption by the frontend, so it is the job of the `listener` function to transform the entities into the desired format.

This is a single function with the single purpose of materializing entities, but there may be many different domain event types to process. The function leverages the *pipeline* feature of the stream processing framework to preserve the maintainability of the logic for the individual event types.

Listener functions are optional. If a BFF service does not need to react to domain events, then it does not need a listener function. However, most BFF services will have a listener function. It is only the BFF services at the very beginning of the data life cycle that may not need a listener function.

The following `serverless.yml` fragment from the BFF service template creates the listener function and configures it to consume from a stream in the event hub:

```
functions:
...
listener:
handler: src/listener/index.handle
events:
- stream:
type: kinesis
arn: ${cf:event-hub-${opt:stage}.streamArn}
```

We covered stream processing in general in *Chapter 4, Trusting Facts and Eventual Consistency*, and we covered the details of creating materialized views in *Chapter 5, Turning the Cloud into the Database.*

Trigger function

The `trigger` function fills the fundamental role in an autonomous service of implementing the *database-first* variation of the Event Sourcing pattern. It is a stream processor that consumes change events from the datastore's CDC stream and publishes domain events to the event hub for consumption by downstream services.

This is a single function with the single purpose of publishing domain events after a user executes a command, but there may be many different change event types to process. The function leverages the *pipeline* feature of the stream processing framework to preserve the maintainability of the logic for the individual event types.

Trigger functions are optional. If a BFF service is read-only (that is, it has no commands), then it does not need a trigger function. BFF services that work toward the end of the data life cycle tend to be read-only. The datastore must support CDC streams; otherwise, we need to use the *stream-first* variant of the Event Sourcing pattern to produce domain events.

The following `serverless.yml` fragment from the BFF service template creates the `trigger` function and configures it to consume from the datastore's CDC stream:

```
functions:
...
trigger:
handler: src/trigger/index.handle
```

```
events:
- stream:
type: dynamodb
arn:
Fn::GetAtt: [ EntitiesTable, StreamArn ]
          ...
```

We covered stream processing in general in *Chapter 4*, *Trusting Facts and Eventual Consistency*, and we covered the details of leveraging a datastore's CDC stream in *Chapter 5*, *Turning the Cloud into the Database*.

Now, let's look at the nano architecture of functions.

Dissecting function-level nano architecture

In *Chapter 2*, *Defining Boundaries and Letting Go*, we introduced the concept of nano (that is, function-level) hexagonal architecture. This architecture level describes how we create a clean structure within a given function so that all internal dependencies are acyclic and we decouple the business logic from external dependencies. To accomplish this, we put some simple layering in place, as shown in the following diagram:

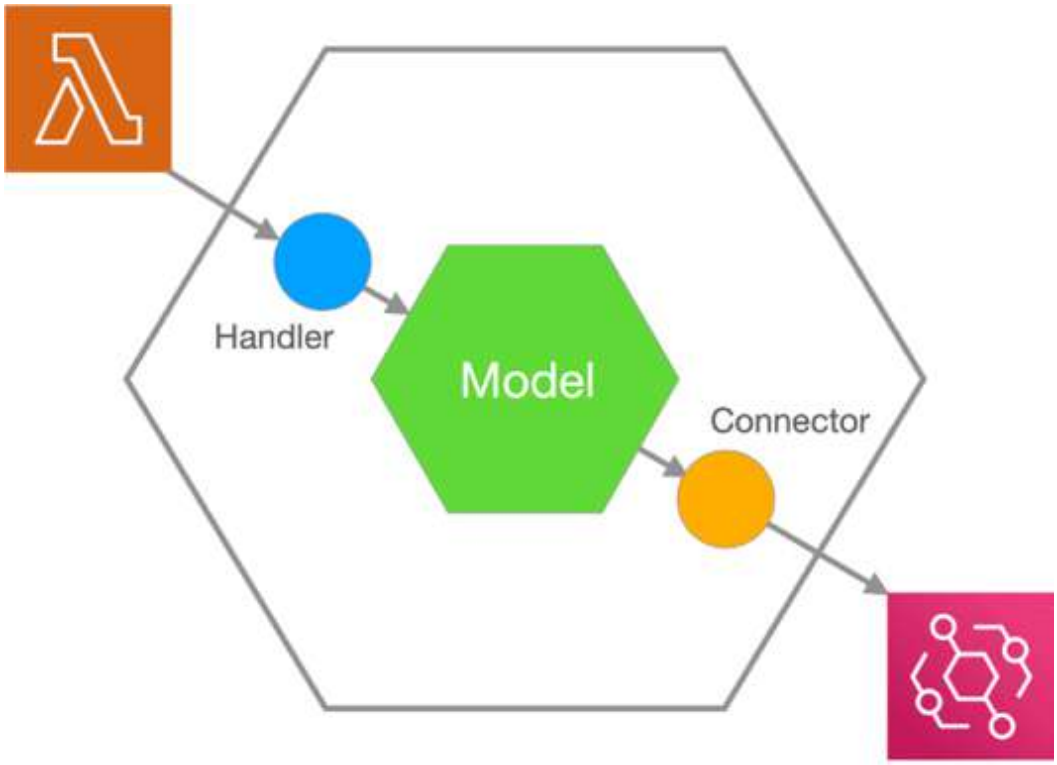


Figure 6.2: Function-level nano-architecture

Akin to Alistair Cockburn's *Hexagonal Architecture* and Robert Martin's *Clean Architecture*, we want to isolate the business logic from the concerns of interacting with the technical resources, such as the datastore and event hub, and the function execution environment. Therefore, we implement the business logic in `Model` classes, wrap all resource calls in `Connector` classes, and hide the details of the execution environment in the `handlers`. This does more than make the code more *portable*; most importantly, it makes it more *testable*. Let's dig into these three layers.

Models

The following example shows the structure of a typical `Model` class:

```
export default class Model {
  constructor( connector ) {
    this.connector = connector;
  }
  save(id, input) { ... };
  getBy(id) {
    return this.connector.get(id);
  };
  queryBy(username) { ... };
  submit(id) { ... };
  delete(id) { ... };
}
export const toUpdateRequest = (uow) => ({ ... });
export const toDomainEvent = (uow) => ({ ... });
```

A BFF service works with a small slice of the system's overarching logical domain model. Each Model class represents a separate domain aggregate used in the BFF service. Each method in a model, such as `save`, `submit`, and `getBy`, corresponds to a command or query provided by the service to the frontend.

A BFF service stores the domain entities in a format best suited for the frontend. The models provide mapper functions, such as `toUpdateRequest` and `toDomainEvent`, which transform the data between the internal and external data formats. These mapper functions support the `listener` and `trigger` functions, respectively.

Connectors

A Model class has a constructor for injecting connectors. The following example shows the structure of a typical `Connector` class:

```
class Connector {
  constructor( tableName ) {
```

```
      this.tableName = tableName;
      this.db = new DynamoDB.DocumentClient({ ... });
    }
    get(id) {
      const params = {
        TableName: this.tableName,
        KeyConditionExpression: '#pk = :id',
        ...
      };
      return this.db.query(params).promise()
        .then((data) => data.Items);
    }
    update(Key, inputParams) { ... }
  }
```

In this specific example, we see a connector that is wrapping the calls to AWS DynamoDB. The `Model` class dictates the signatures of the `Connector` methods, and the connector encapsulates the details of interacting with the resource. This will allow the model to work with different connector implementations. For example, during the early stages of development, it may be helpful to implement a connector that holds data in memory, so that we can quickly experiment with the functionality. Then, we plug in the real connector when we are more confident with the requirements.

The connectors also make it easier to write unit tests that execute in isolation, because it is easier to write mocks for connectors than it is to write mocks for resources. Developers only need to write a small set of connector unit tests that have the complexity of mocking the resources. Then, for the model unit tests, which account for the majority of test cases, it is only necessary to mock the connectors.

Each connector defines a constructor with configuration options. This provides a clean separation between the connector and the execution environment. For example, at runtime, a function will receive connection

information in environment variables, whereas unit tests will simply assign test values.

Handlers

Models are agnostic to the execution environment. Instead, we use a handler to wrap and adapt a model to the given environment. For example, we need to execute the model in the chosen runtime environment, but we also need to test the model independently. Plus, we need to invoke different parts of the model in our listener, trigger, and command/query functions, so each will have a different handler.

We perform *dependency injection* in the handlers. We use constructor-based dependency injection because it is simple and lightweight and, therefore, does not have a negative impact on cold-start times. We will see examples of this in the next section. The handler also initializes the models and connectors with environment variables and context variables.

We should be able to implement a BFF service using REST or GraphQL without having to change the Model and Connector classes. Now, let’s look at using REST versus GraphQL and how these interact with the models and connectors.

Choosing between REST and GraphQL

GraphQL has become very popular in recent years, which raises the question of whether it is better to use REST or GraphQL. The short answer is that it depends on the user activity.

Does a user activity involve many users interacting with the same data or a few users interacting with different data? Taking our food delivery system as an example, a few employees from each restaurant will interact with just the menu of their restaurant, whereas many customers will interact with the menus of many restaurants. Furthermore, the customers are only reading the menus, and the restaurant employees are editing the menus.

This means that we can use the `cache-control` header to improve the responsiveness of the system for customers and reduce the cost of the system. This is where *REST* excels. **Representational State Transfer** (**REST**) was designed to take full advantage of the infrastructure that makes up the internet, such as routers and content delivery networks. So, it makes sense to use REST for a customer-facing menu service.

GraphQL, on the other hand, typically uses the `POST` operation, so it is not well suited for read-only scenarios. However, it is very well suited for more ad hoc activities where the user is navigating through and editing data, such as restaurant employees maintaining menus or customers maintaining their account information and user preferences.

In *Chapter 7*, *Bridging Intersystem Gaps*, we will cover the topic of *webhooks*, which usually require RESTful endpoints. We will also cover the topic of open interfaces, which traditionally expose a RESTful endpoint. These are akin to a BFF service, but they support an unbounded number of external systems. This means that versioning and backward compatibility are of utmost importance. Backward compatibility is an area where GraphQL excels because clients declare the exact fields they need. This makes it easier to track how clients use the interface and craft backward-compatible changes.

A major advantage of GraphQL is that its client-driven query approach tends to be less chatty than REST. A typical RESTful interface will require an application to make several calls to retrieve a full domain aggregate. This kind of lazy retrieval may be preferred if the user does not need the full object. GraphQL, on the other hand, lets the client specify the fields to return and the associations to traverse, which allows the client to retrieve more data all at once. This level of flexibility is less important for a highly specialized BFF service, but it is very helpful for a generalized open interface.

My general guideline is to use REST for BFF services and GraphQL for Open APIs. Sometimes, it may be necessary to support both. Let's see how models and connectors help us do this.

REST

REST is the traditional choice for implementing services. As we have discussed, it is better for read-only services, webhooks, and BFF services. The following code fragments implement the `handler` for the `rest` function we declared in the *Dissecting the Backend for Frontend pattern* section:

```
const api = require('lambda-api')();
import Connector from '../connectors/dynamodb';
import Thing from '../models/thing';
const connector = new
Connector(process.env.ENTITY_TABLE_NAME);
api.app({
  models: {
    thing: new Thing(connector),
  }
});
...
export const handle = async (req, ctx) =>
  api.run(req, ctx);
```

The handler uses the `lambda-api` framework, which is very similar to Express, but with a significantly smaller footprint. It works with both AWS API Gateway and AWS Application Load Balancer and hides the differences. The handler function simply needs to wire the various pieces together. First, we register the `models` with the framework and inject the connectors.

Next, we implement the `route` that will handle requests to retrieve `Thing` entities by their identifier:

```
api.get('/things/:id', (request, response) =>
  request.namespace.models.thing
  .get(request.params.id)
  .then((data) => response.status(200).json(data)));
```

The `route` function has access to the `models` through the `request` object. The model handles the business logic, and the connector interacts with the datastore. This means the `route` function only needs to implement the glue code to map the inputs from the `request` object to the `model` and map the output from the `model` to the `response` object.

Now let's look at how we can use the same model and connector in a GraphQL function.

GraphQL

GraphQL is a popular choice for implementing services. As we have discussed, it is best for implementing open interfaces, because client-driven queries are naturally backward compatible, and it is self-documenting.

Here is an example of hooking up GraphQL in a lambda function. The following `serverless.yml` fragment creates an API gateway for the service:

```
functions:
graphql:
handler: src/graphql/index.handle
events:
- http:
path: graphql
method: post
```

There is one route that directs `POST` requests for the `graphql` path to the function that implements a GraphQL schema. Next, we have a fragment of a GraphQL schema that defines the `Thing` domain entity and a query to retrieve `Thing` by ID:

```
 type Thing {
    id: String!
name: String
description: String
  }
  extend type Query {
    thing(id: String!): Thing
  }
```

This schema corresponds to the `Model` class we defined in the *Models* section. There will typically be a one-to-one relationship between schema fragments like this one and `Model` classes. We will see how resolvers bring the two together in a moment. Next, we implement the `handler` for the `graphql` function:

```
import { ApolloServer, gql } from 'apollo-server-lambda';
import schema from './schema';
import Connector from '../connectors/dynamodb';
import Thing from '../models/thing';
const connector = new
Connector(process.env.ENTITY_TABLE_NAME);
const handler = new ApolloServer({
```

```
    ...schema,
    context: () => ({
      models: {
        thing: new Thing(connector),
      },
    }),
  }).createHandler();
  export const handle = (req, ctx, callback) =>
    handler(req, ctx, callback);
```

The handler uses the `Apollo` framework to do the heavy lifting of processing GraphQL requests. The handler function simply needs to wire the various pieces together. Once again, we register the `models` and inject the connectors. We also register the GraphQL `schema`. Finally, we implement the `resolvers` that map the schema to the `models`:

```
  export default const resolvers = {
    Query: {
      thing(_, { id }, context) {
        return context.models.thing.get(id);
      },
      ...
    },
    Mutation: { ... },
  };
```

In this case, we see the resolver that handles queries for `Thing` entities by their identifier. The resolver has access to the `models` through the `context` object. The model handles the business logic, and the connector interacts with the datastore. This means the resolver only needs to implement the glue code to map the inputs from the query to the model and return the output from the model.

These examples demonstrate how we use handlers to decouple the models from the execution environment. Business logic lives in the model classes

and the connector classes encapsulate the external dependencies. Then, we glue everything together in the execution environment with the appropriate framework.

Now that we have seen some of the lower-level details, let's see how we can put all this to work in applications with some different kinds of BFF services.

Implementing different kinds of BFF services

Up to this point, we have learned how to control the scope of a BFF service and we have seen its major building blocks. In *Chapter 5, Turning the Cloud into the Database*, we discussed how different actors interact with data in different ways as it moves through its life cycle. Let's look at some different kinds of BFF services that cater to the different phases of the data life cycle. These include the **CRUD**, **Lov**, **Task**, **Search**, **Action**, **Dashboard**, **Reporting**, and **Archive** BFF services.

CRUD BFF services

A **CRUD BFF** service is the most general variation of the BFF service pattern. These BFF services let us **Create, Read, Update, and Delete** (**CRUD**) data. They support user activities in the Create phase of the data life cycle.

For example, the Restaurant BFF service in our food delivery system allows restaurant owners and representatives to create and maintain their menus. The events from this service will feed the Menu BFF services, which we will cover shortly in the *Search BFF services* section. We often refer to the data

in these services as *reference data* or *master data* that supports all the downstream business processes. We also refer to these services as the *system of record* for this data. We can also think of this as *slow data* because it changes infrequently.

Figure 6.1 in the *Dissecting the Backend for Frontend pattern* section depicts the resources involved in a CRUD BFF service. These BFF services may not need a listener function, since we primarily use them to create data that we utilize downstream. However, in an evolutionary system, these services may replace similar functionality in a legacy system. So, we may need a listener function to stay in sync with any data changes in the legacy system. We cover this topic in *Chapter 7, Bridging Intersystem Gaps*.

List-of-values (Lov) BFF services

An **Lov BFF** service is a variation of the BFF service pattern that provides efficient, read-only access to data that we need to support data entry user activities.

For example, a data entry form usually has drop-down lists that give the user the valid values for a given field in the form. Most of these lists are static, but many lists need access to dynamic data. This is an example of data in the *Use* phase of the data life cycle. The dynamic data was created upstream in a separate user activity in a BFF service that we optimized for writing data. Now we are using the data in a different downstream user activity, and we want to access the data from a BFF service that we optimize for these kinds of reads.

The following figure depicts the resources involved in an Lov BFF service:

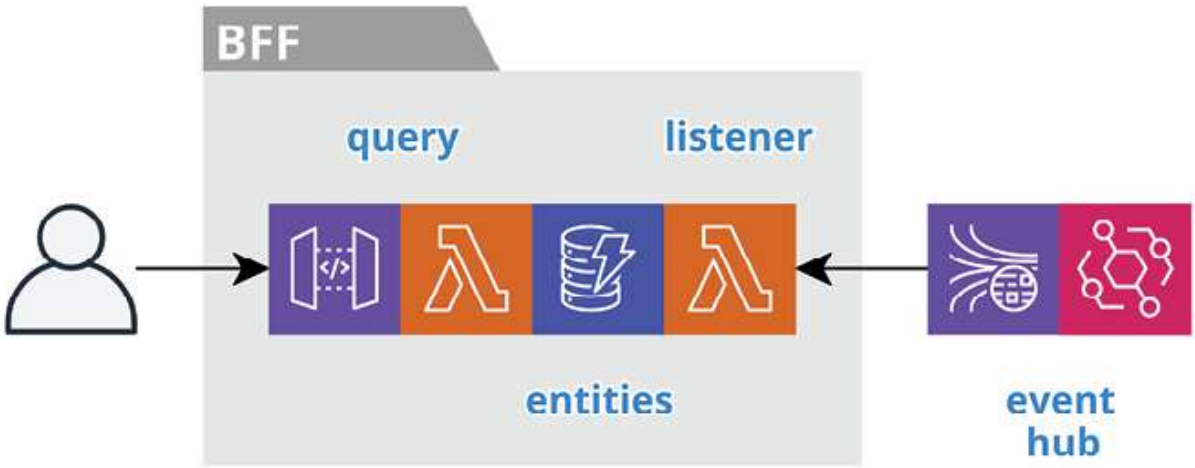


Figure 6.3: Lov BFF service

An Lov service is a read-only BFF, so it does not have `command` or `trigger` functions. It only has a `listener` function and a `query` function. The `listener` function consumes `created`, `updated`, and `deleted` event types for various domain entity types and materializes the data in the `entities` table. We only store the data that we need to populate the drop-down lists, such as a code/identifier and a human-readable description, and maybe some flags we can use to filter the lists. We will ignore any other data in the domain entity to keep the data in this service lean.

Following the *Single-Table design pattern*, we will use the code as the `pk`, and we will set the `discriminator` field on each row to identify the domain entity type. The `query` function will use an index on the discriminator so that we can easily and efficiently access the list of values for a specific domain entity type. For large datasets, we can include a filter when we call the `query` function.

Each autonomous subsystem will typically have a single Lov service. We referred to this BFF variation in the *Designing for offline-first* section of *Chapter 3, Taming the Presentation Tier*. These services simplify the

frontend because we can retrieve all the necessary reference data from a single service on startup and cache it so that it is available offline.

Task BFF services

A **Task BFF service** is a variation of the BFF service pattern that supports a step (that is, a task) in a larger business process.

Tasks are the workhorses of BFF services, as they constitute most BFF services. A Task BFF provides a well-defined query to access the data the user needs to perform the specific task. It also provides the set of commands the user needs to accomplish the task, including a command to signal when the task is complete.

For example, from our food delivery system, the Delivery BFF allows drivers to acquire a delivery assignment, access the delivery information, and indicate when the delivery is complete. The Order BFF provides restaurants with a list of outstanding orders, so they can prepare the food for delivery. The Order Status BFF allows customers to track their orders. These are three different views on the same business process, but we design them for different users with different activities.

Figure 6.1 in the *Dissecting the Backend for Frontend pattern* section depicts the resources involved in a Task BFF service.

A Task BFF implements a step in a larger business process. The `listener` function consumes domain events that initiate the tasks. It materializes the data so users can query their work. These domain events may come from control services that orchestrate the business processes or the individual services may choreograph the business processes themselves. We cover orchestration and choreography in *Chapter 8, Reacting to Events with More*

Events. The `trigger` function produces domain events to record the outcome of any intermediate commands, and it ultimately produces the domain event that signals the completion of a task and moves the larger business process forward.

Search BFF services

A **Search BFF** service is a variation of the BFF service pattern that provides read-only access to the data produced by one or more upstream services.

Whereas a Task BFF service provides its own well-defined queries, a Search BFF service helps users narrow in on what they are looking for. The users may not know exactly what they are looking for or even whether the information exists. They may only have a general idea of what they want, and they are looking to see what is available.

For example, the Menu BFF service in the Customer autonomous subsystem of our food delivery system provides the ability to search for restaurants and menu items by name, category, location, and more. The Restaurant BFF CRUD service that we covered earlier allows restaurants to maintain their menus, but the Menu BFF search service provides the high performance and scalability needed to support millions of customers. In the back office, a search service would provide a central location for employees to search for all the business's domain information, such as customers and cases in a customer support system.

The following figure depicts the resources involved in a Search BFF service:

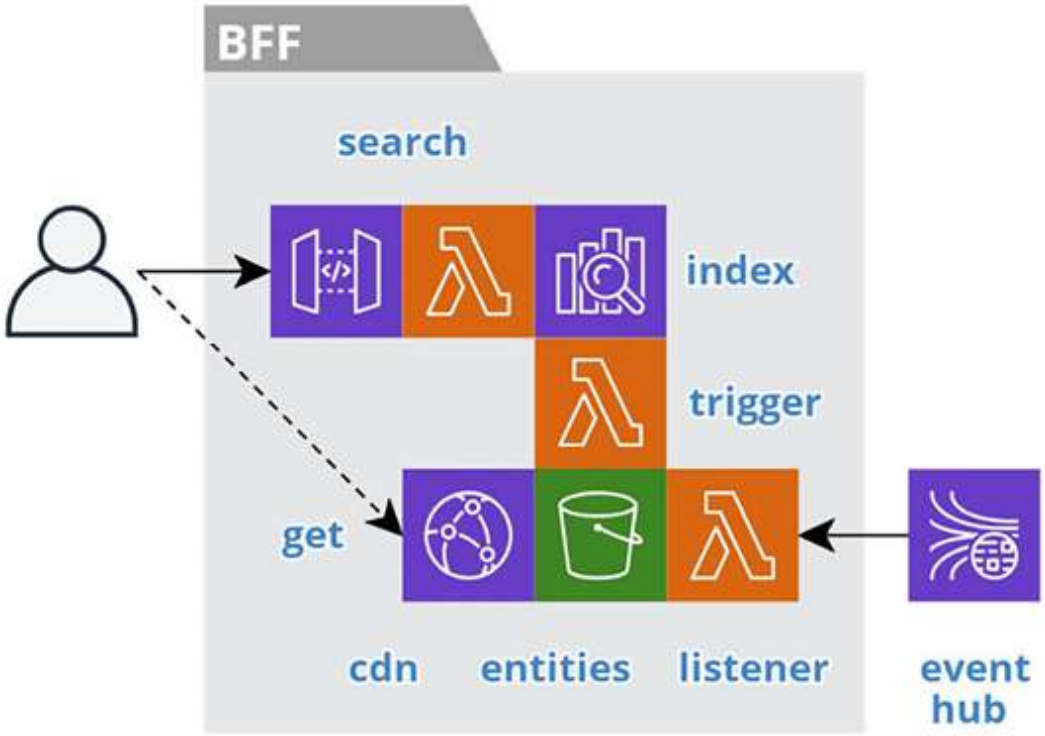


Figure 6.4: Search BFF service

At the heart of a search service is an `index`, such as **Elasticsearch**. The `search` interface provides access to the index through an API gateway and a search function. Each search result contains summary information and the URL to access the detailed entity information.

The domain `entities` data is the perfect candidate for object storage, such as an AWS S3 bucket, because the volume of data is unbounded and it is read-only, reasonably static, and only accessed by the identifier. We store the data in the JSON format, which we will return to the client, along with any images. The object keys should follow a good RESTful structure, such as `/restaurants/123/menuitems/987`.

For public information, like the Menu service, we serve the domain entities from the edge of the cloud using a **Content Delivery Network** (**CDN**), such as AWS CloudFront. We accomplish this by setting the `cache-control`

property on the objects in storage so that we can cache the requests on the browser and the CDN. This provides the low latency and massive scale needed to support an unbounded number of customers. We discussed the origins of this approach in the *Turning the database inside out* section of *Chapter 5, Turning the Cloud into the Database*. For private data, such as a back-office search service, the client will access the data through a signed URL within a short TTL.

The `listener` function is responsible for materializing the entities and images in the S3 bucket. Then, the `trigger` function indexes the data, only after it is available in the bucket. Chaining these steps is important because each step is atomic, as we discussed in *Chapter 4, Trusting Facts and Eventual Consistency*. Splitting these steps ensures that the index will not return a result with a broken link.

Action BFF services

An **Action BFF** service is a variation of the BFF service pattern that performs a specific action in the context of another BFF service.

For example, the Menu BFF service in our food delivery system is a Search BFF that supports browsing and navigating restaurants and menu items, but it does not support adding items to a cart. This is the job of an Action BFF, such as a cart service. This action BFF supports adding items to a cart and stores the menu item's identifier, the quantity, and any configured options. Once the customer submits the order, the cart service no longer needs to retain the cart information because it is not editable, and the `order-submitted` domain event is a record of the state.

The following figure depicts the resources involved in an Action BFF service:

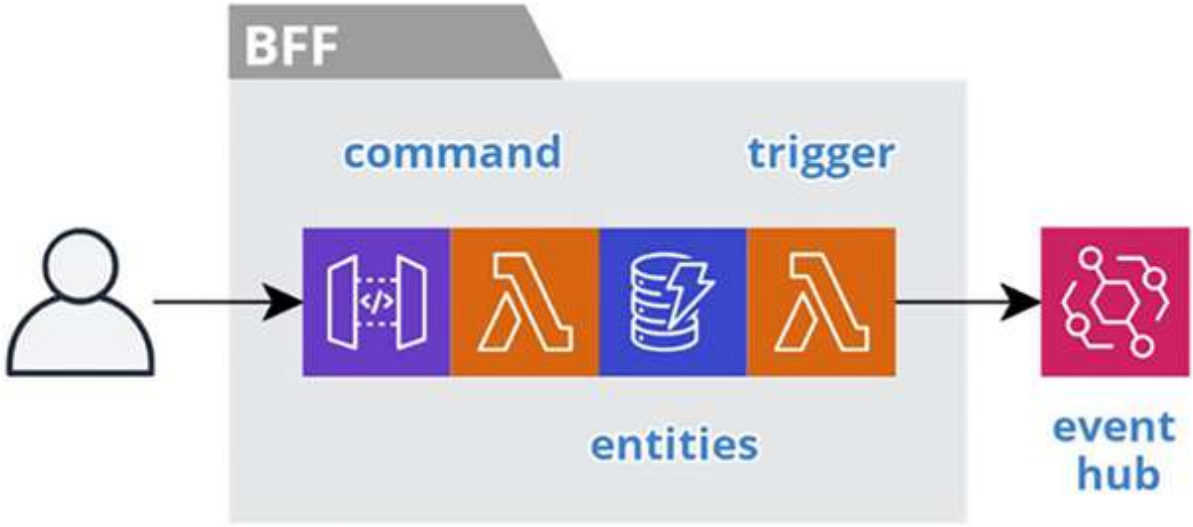


Figure 6.5: Action BFF service

An Action service is responsible for performing a set of commands. We can think of these services as headless because they need other services to provide context for the inputs to a command. In other words, we pass all the necessary information into a command. They have no listener to create materialized views that the service can query to provide its own context. An Action service may have a query to return the results of its own actions, but the service does not retain this information for any longer than necessary. Finally, a `trigger` function produces domain events to record the outcome of the commands.

Dashboard BFF services

A **Dashboard BFF** service is a variation of the BFF service pattern that provides users with a read-only, running tally of important domain values.

For example, a banking application will provide users with a dashboard of all their balances. In our food delivery system, restaurants can see statistics on how many customers viewed their menus, the top viewed items, and

order statistics, so that they can gauge the value of using the food delivery service.

The following figure depicts the resources involved in a Dashboard BFF service:

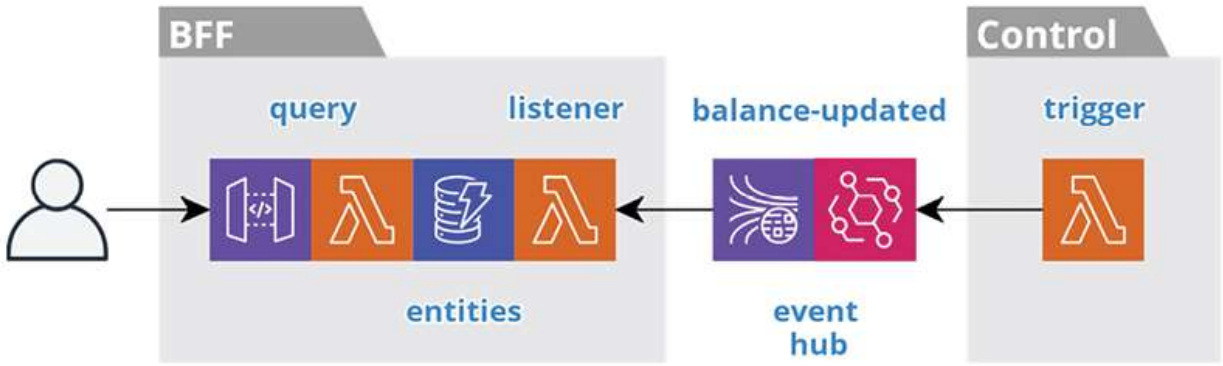


Figure 6.6: Dashboard BFF service

These user activities are typically externally facing and, therefore, require fast response times and high availability. These requirements, combined with the fact that the user experience is not ad hoc, mean that these services do not need and should not rely on complex analytics infrastructure.

Instead, a Dashboard BFF service works with a control service that leverages ACID 2.0 transactions to perform the necessary calculations, as we will cover in the *Calculating event sourcing snapshots* section in *Chapter 8*, *Reacting to Events with More Events*. The control service consumes lower-order domain events, calculates the current value, and produces a higher-order domain event, such as `balance-updated`.

This leaves the BFF service with the simple responsibility of delivering the read-only information to the user. Dashboard user activities have a well-defined access pattern. For example, each record contains a specific value, with `pk` set to the user or group identifier, and `sk` set to the value's identifier.

The `listener` function consumes the higher-order events and materializes the values. The `query` simply returns the set of pre-calculated values for the current user. As the Dashboard service is autonomous, it can always return the most recently received pre-calculated values.

Reporting BFF services

A **Reporting BFF** service is a variation of the BFF service pattern that provides users with a historical view of the data in the form of periodic reports.

In *Chapter 5, Turning the Cloud into the Database*, we learned how to control the volume of data and keep data lean in the individual services. Services set a TTL on each record so that data expires when they no longer need it. However, it is common to have reporting requirements on historical data. This is a different user activity, for a different actor, in a later phase of the data life cycle. So, we create Reporting services downstream for these activities.

The following figure depicts the resources involved in a Reporting BFF service:

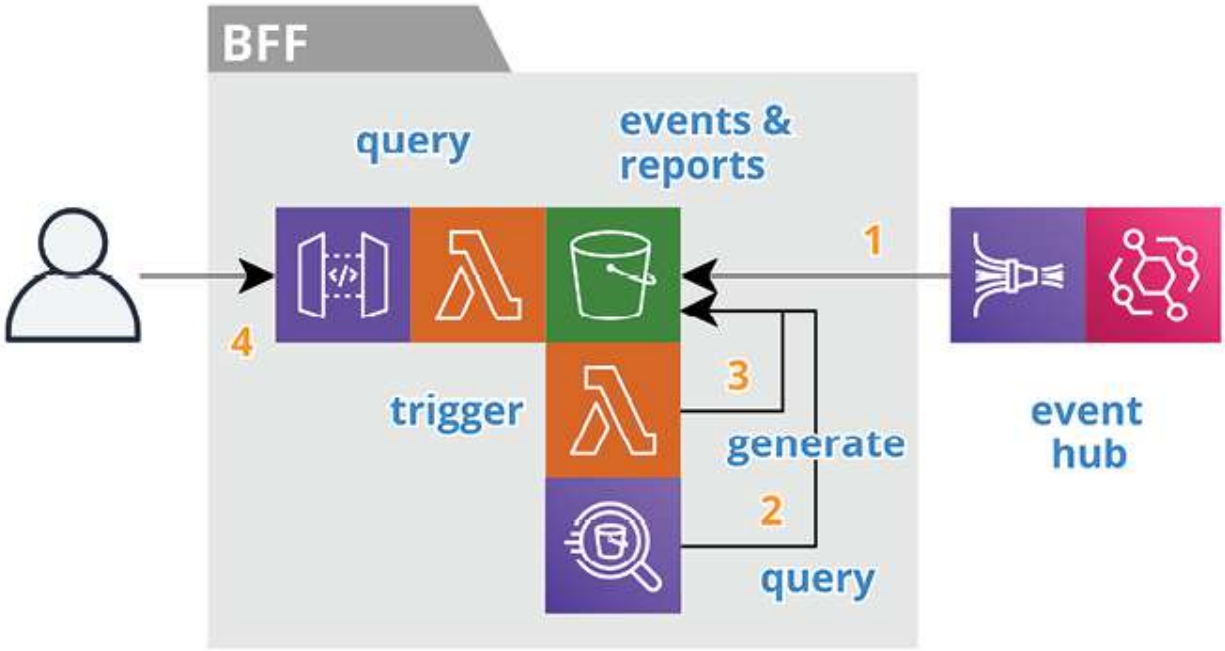


Figure 6.7: Reporting BFF service

On a periodic basis, an actor expects to have access to reports, such as a monthly report on their area of responsibility. Traditionally, this data lives in a monolithic relational database and the users execute jobs to generate the reports and then wait for the jobs to complete. Instead, we can continuously prepare the reports as new data flows in so that they are ready at the desired frequency.

We organize, version, and store the generated reports in an S3 bucket with an appropriate key structure, such as `/division/yyyy/mm/dd/report-name.pdf`. The `query` interface allows the user to browse the reports. The sequence follows:

1. First, we consume domain events through a channel, such as AWS Firehose, which stores the events in S3 in a suitable format.
2. Next, the trigger function invokes a service, like AWS Athena, to query the events and write the results back to the bucket.

3. This in turn causes the trigger function to generate the report from the query results and store it in the bucket.
4. Finally, at any time, the user can see the latest version of a report with the most recent data.

The events in the bucket have a TTL that is short enough to control the size of the bucket but long enough for late-arriving data to generate a new version of the report. We can retain the reports themselves indefinitely and age them into cold storage.

Archive BFF services

An Archive BFF service is a variation of the BFF service pattern that serves as the ultimate data collector at the end of the data life cycle. These services are governed by record management requirements that specify how long a company must retain various pieces of data.

In *Chapter 5, Turning the Cloud into the Database*, we learned about the importance of *proactive archiving* and how it enables services to keep their data lean. Services set a TTL on each record so that they remove data when they no longer need it. Archive BFF services collect all events, which allows other services to remain lean.

The following figure depicts the resources involved in an Archive BFF service:

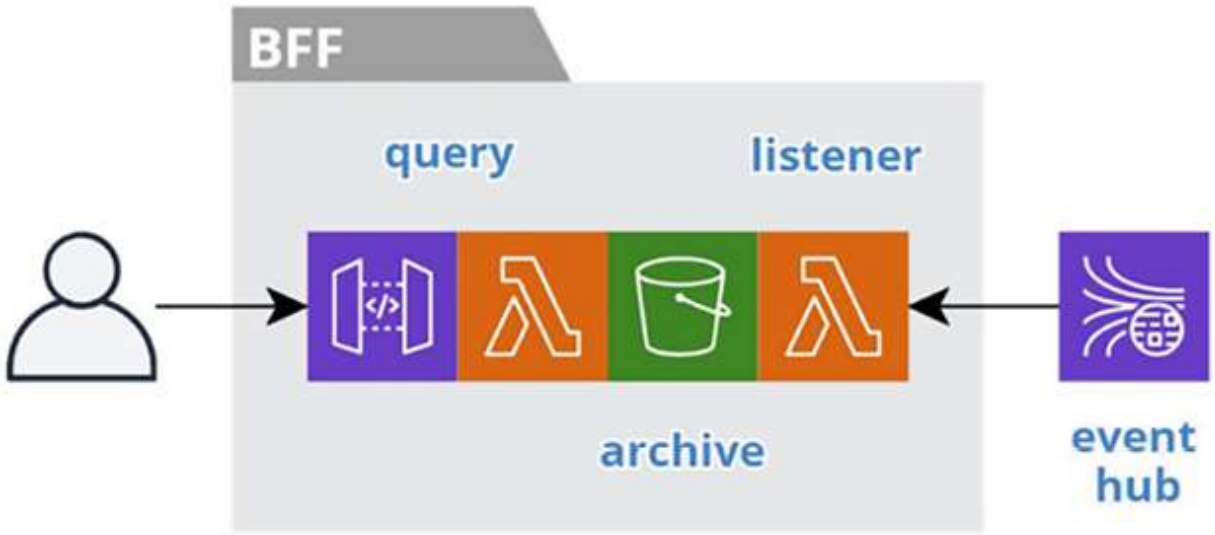


Figure 6.8: Archiving service

Archive BFF services are proactive because they do not wait to collect data at the end of the life cycle. They continuously collect data from the moment users create it so that upstream services can delete data freely.

The `listener` function consumes domain events and stores the data in an S3 bucket. This is akin to an event lake, but we organize the data around a specific domain entity identifier, such as customer, restaurant, or driver. A detailed history accumulates, which we can use in the future for auditing and restoration. The `query` interface allows a user to browse the archive. The data ages to cold storage and we purge it after a regulated time period.

To put the Archive service into context, we can use a case management system as an example. A Task BFF service is responsible for active cases. While a case is active, the index in the Search BFF service points to the data in the Task BFF service. When a case officer closes a case, the Task BFF service deletes its copy of the case data, and we update the search index to point to the case data that has accumulated in the Archive BFF service. If we

need to reopen a case, then we restore the case data from the archive to the Task BFF service and update the search index.

Each autonomous subsystem should have an archiving service for its domain aggregates so that its services remain lean. If a subsystem has many domain aggregates, then it may warrant multiple archives. As a whole, a system should have an archive subsystem at the end of the data life cycle that is responsible to the regulatory actor. This allows the other autonomous subsystems to remain lean.

Summary

In this chapter, we built on the patterns we introduced in earlier chapters and you learned why it is important to create services that focus on user activities and how this helps teams accelerate the pace of innovation. You learned about the BFF pattern and how to combine the Event Sourcing and CQRS patterns to create user-facing services. You learned how to apply the BFF pattern throughout the different phases of the data life cycle. You also learned how to implement a function-level nano architecture and how to choose between REST and GraphQL.

In the next chapter, we will cover the **External Service Gateway** (**ESG**) pattern and see how it creates an anti-corruption layer that frees teams to focus on the functionality of their system. We will look at variations of the ESG pattern and cover integration with third-party systems, legacy systems, other autonomous subsystems, and more.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/B8TubSxc35

7

Bridging Intersystem Gaps

In *Chapter 6, A Best Friend for the Frontend*, we covered the Backend for Frontend autonomous service pattern. These services work at the boundary of the system to support end users. They cater to the kinds of changes driven by human actors. In this chapter, we turn our attention to integrating with external systems. We will create boundary services that cater to the kinds of changes driven by these external system actors so that external concerns and changes do not corrupt your system.

In this chapter, we're going to cover the following main topics:

- Creating an anti-corruption layer
- Dissecting the External Service Gateway pattern
- Integrating with third-party systems
- Integrating with other subsystems
- Integrating across cloud providers
- Integrating with legacy systems
- Providing an open API and SPI
- Tackling common data challenges

Creating an anti-corruption layer

In *Chapter 2, Defining Boundaries and Letting Go*, we highlighted the role that the **Single Responsibility Principle** (**SRP**) plays in helping to define a

system’s boundaries. Each bounded context should be responsible to one and only one actor. In other words, it is the actors who interact with a system that are the driving force behind any reason to change the system. So, we draw boundaries around the interactions of the various actors to limit the impact of any changes they will cause.

We identified *three autonomous service patterns* that each cater to a different type of *actor*. In this chapter, we dig into the details of the **External Service Gateway** (**ESG**) pattern. The ESG pattern works at the boundary of the system to provide an *anti-corruption layer* that encapsulates the details of interacting with other systems, such as third-party, legacy, and sister subsystems. These ESG services act as a bridge to exchange events with the external systems.

External systems are owned by a different organization. This may be another organization in your company or another company altogether. The implication is that we have little to no control over the design and architecture of these systems, nor do we have much say over when and how they might change. This means it is in our best interest to isolate the interactions with external systems, so that we protect the rest of our system, and allow each to change independently.

The term *anti-corruption layer* was popularized by Eric Evans in his book *Domain-Driven Design*. The Gang of Four institutionalized the *Adapter* pattern in their pivotal book *Design Patterns: Elements of Reusable Object-Oriented Software*. Robert Martin recommends an *Interface Adapter* layer in his book *Clean Architecture*, and in Alistair Cockburn’s hexagonal architecture, core components expose *ports* and *adapters* glue them to the outside world. The overarching idea across all of these is to separate and decouple external concerns from internal concerns by creating an isolation

layer. Our ESG services will create an anti-corruption layer at the macro (subsystem) level.

Macro-level ports and adapters

The ESG pattern helps us build an anti-corruption layer at the macro or subsystem level, in a system of autonomous services. As we covered in *Chapter 6, A Best Friend for the Frontend*, each autonomous service (that is, BFF, ESG, or Control) employs layers of classes within, to promote the clean, decoupled code that allows us to easily test services in isolation from external resources. But our focus here is on how ESG services themselves act as an isolation or anti-corruption layer for the system, from external systems. Borrowing from the hexagonal architecture, we can think of domain events as **ports** and ESG services as **adapters**.

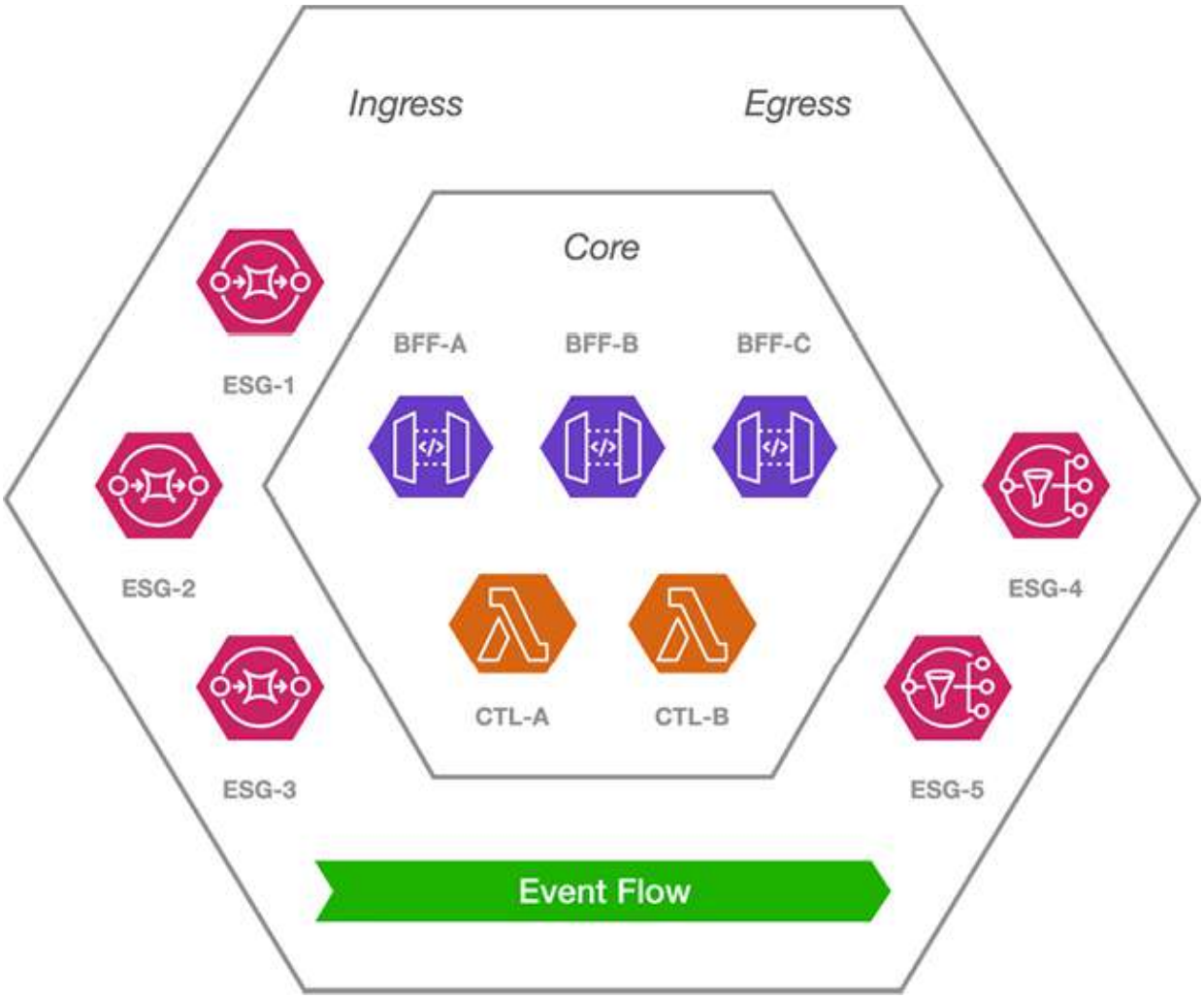


Figure 7.1: Autonomous subsystem – Subsystem ESG ingress and egress layer

In *Figure 7.1*, we see a hexagonal representation of a typical autonomous subsystem with driving (ingress) services on the left and driven (egress) services on the right. The relative position of services from left (upstream) to right (downstream) implies the flow of events on a timeline. Events flow in from upstream systems via ingress ESGs; the core BFF and Control services of the subsystem make their contributions and produce more events. Events flow out to downstream systems via egress ESGs. The ESGs completely decouple the core of the subsystem from the external systems.

Substitution principle

Each ESG service is focused on adapting a single external system to the internal model of the system. Each ESG is a separately deployable autonomous service. This allows us to employ the **Liskov Substitution Principle** (**LSP**). For example, over time, there may be a business reason to switch from one third-party system to another, and we can simply replace one ESG with another. Or with legacy migration and the use of the *Strangler pattern*, we can leverage the ESG pattern to run the old and new simultaneously and simply turn off the old when it is no longer needed, without upstream or downstream impacts. We will dig into the Strangler pattern in the *Integrating with legacy systems* section.

External bulkhead

ESG services act as a *bulkhead* as well. They protect systems from failures in each other. For example, if an external system is experiencing an outage, then an ESG can leverage a *circuit breaker* to minimize the impact on our system. Conversely, the natural *backpressure* of event stream processing can prevent our system from overpowering a less robust external system. In either case, the volatility and/or limitations of an external system are absorbed by its corresponding ESG service, because the ESG addresses any impedance mismatch between the two systems.

Let’s take a look at the anatomy of an ESG service before we dive into various applications of the pattern.

Dissecting the External Service Gateway pattern

In *Chapter 2, Defining Boundaries and Letting Go*, we did some event storming to identify a system's domain events, human actors, and external system actors. We also divided the system into autonomous subsystems. Now, we need to get our bearings straight and clarify which external systems sit upstream and which sit downstream of each subsystem. This is where subsystem-level hexagonal diagrams like *Figure 7.1* come in handy.

For each external system, we add an ESG service to a subsystem's diagram on the *driving* or *driven* side to indicate whether events will flow in from the external system or flow out to the external system. If events flow in both directions to a given external system, then we will likely want separate ingress and egress ESGs for that external system. We also need to articulate which domain events the ESG services will produce and consume.

From here, we can start to dive down into the details of the individual ESGs. First, we need to understand the available options for connecting to the external system, because this will drive a lot of our implementation decisions. Then, we need to define the semantic mappings between the external and internal domain entities that we will implement in the ingress and egress gateways. We also need to decide how to package and where to deploy the ESG service. Let's dig into each of these topics.

Connectivity

How we connect to an external system is driven almost entirely by the type of external interface it provides. Does the system support a real-time interface or only a batch interface? Is the external interface synchronous or asynchronous? What protocol does it support, such as REST, SOAP, JMS, AMQP, or so forth?

Is the payload formatted in JSON, XML, or something else? How is the external interface secured? Is it owned by a third party or is it a legacy system? Is it running on-premises or in the cloud?

All these questions, and more, are the kinds of challenges an ESG handles and hides so that these details and differences do not penetrate and corrupt the rest of your system. Throughout this chapter, we will flesh out variations on the ESG pattern that account for many of these scenarios. We will address security topics in *Chapter 10, Securing Autonomous Subsystems in Depth*.

Semantic transformation

The primary function of an ESG is to perform the semantic transformations between the internal and external data formats. All the details of connectivity may be a challenge upfront, but the majority of the business logic and churn in requirements will revolve around the semantic transformations. An external system that works with the same logic data will most likely model it in different formats, using different naming conventions, having more or fewer fields, and possibly having different cardinalities for associated data. We will see some transformation examples shortly in the *Egress* and *Ingress* sections.

As we discussed in *Chapter 2, Defining Boundaries and Letting Go*, our domain events form the system's contracts. By transforming and adapting to these contrasts, the ESG pattern enables the system to evolve as the requirements for and by the external systems change. This is the LSP at work, allowing us to substitute different ESG services as business needs change.

Sometimes, the information needed to perform the semantic transformations is not readily available in the message payloads themselves. In this case, the

ESG will handle enriching the data. We will cover an approach for this later in the chapter, in the *Tackling common data challenges* section.

We will also leverage the event lake to keep an audit trail of what was sent to and received from an external system. As we saw in *Chapter 4, Trusting Facts and Eventual Consistency*, the **raw** field in the standardized event envelope can be used to store the external payloads to ensure that there is no loss of information after any transformations.

Now let's see some example transformations at work in ingress and egress flows.

Ingress

An ingress flow handles integration with an external system to receive external domain events and transform them into internal domain events so that our system can react as needed. In the following example, the external system emits events, and we simply transform and forward the logical domain event from one system to the other. *Figure 7.2* depicts the resources involved:

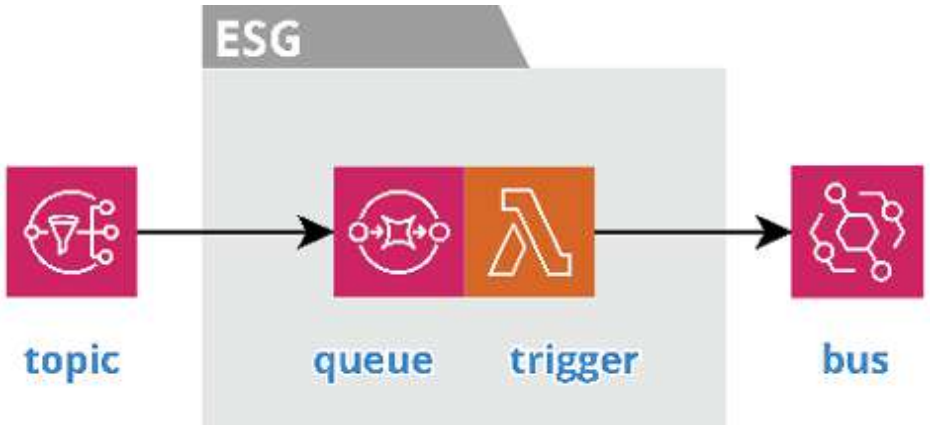


Figure 7.2: ESG – ingress

Below is the sample code for the `trigger` function. It implements a stream processor, such as the ones we covered in *Chapter 4, Trusting Facts and Eventual Consistency*:

```
export const handler = async (event) =>
  fromSqs(event)
    .map(toInternalFormat)
    .through(publish())
    .through(toPromise);
const toInternalFormat = (uow) => ({
  ...uow,
  event: {
    id: uow.record.messageId,
    type: `thing-${uow.record.body.action}`,
    timestamp: uow.record.SentTimestamp,
    thing: {
      field1: uow.record.body.f1,
      field2: uow.record.body.f2,
      field3: uow.record.body.f3,
    },
  },
});
```

The external system emits events on an AWS Simple Notification Service (SNS) **topic**. The ESG creates an AWS Simple Queue Service (SQS) **queue** to *connect* to the external system and receive events from the topic. The **trigger** function consumes messages from the queue. The `toInternalFormat` step *transforms* the external format to the internal event format. The **publish** step puts the internal domain event on the **bus** so that core services can react to the event.

Egress

An egress flow reacts to an internal domain event and handles integration with an external system. In the following example, the external system can

receive events and we simply transform and forward the logical domain event from one system to the other. *Figure 7.3* depicts the resources involved:

Figure 7.3: ESG – egress

Below is the sample code for the `listener` function. It implements a stream processor, such as the ones we covered in *Chapter 4*, *Trusting Facts and Eventual Consistency*:

```
export const handler = async (event) =>
  fromKinesis(event)
    .filter(onEventType)
    .map(toExternalFormat)
    .through(send())
    .through(toPromise);
const onEventType = uow => uow.event.type.match(/thing-*/);
const toExternalFormat = (uow) => ({
  ...uow,
  sendRequest: {
    TopicArn: process.env.TOPIC_ARN,
    Message: JSON.stringify({
      f1: uow.event.thing.field1,
      f2: uow.event.thing.field2,
      f3: uow.event.thing.field3,
    }),
```

```
    },
  });
```

The domain events of interest are available on a `stream` in the event hub. The `listener` function consumes from the stream and filters for the event types of interest, such as all events for the `thing` domain entity. The `toExternalFormat` step *transforms* the internal format to the external message format. The `send` step *connects* to the external system and sends the message.

Packaging

Each ESG lives in its own source code repository and cloud stack, like all autonomous services, as we covered in *Chapter 2, Defining Boundaries and Letting Go*. As a default starting point, there should be one ESG per external system that contains all the ingress and egress pipelines.

However, we may want to create separate ingress and egress ESGs for a given external system. This is useful when the external system uses significantly different technology for its inbound and outbound interfaces.

For large and complex external systems, it can be advantageous to create multiple ESG services that group related pipelines and that change together, such as per external system capability. We usually also combine this approach with creating separate ingress and egress ESGs.

Use a service naming convention that makes it easy to recognize the purpose of the specific ESG. A good convention will alphabetically group related services in the source code tool and cloud console, such as `<external-system-name>-ingress|egress-<capability-name>-gateway`. In our food delivery example, we might have a `stripe-gateway` for processing payments

and a `restaurant-egress-gateway` that provides an open API webhook. For a legacy integration, we might have `legacy-ingress-customer-gateway` and `legacy-ingress-driver-gateway`.

Separate cloud accounts

Most ESGs, such as third-party integrations, fit nicely into the functional flows of a particular autonomous subsystem. All autonomous subsystems should have ESGs for inter-subsystem integration. However, some ESGs are best isolated in their own autonomous subsystems with their own cloud accounts.

Legacy system integrations are candidates for their own subsystems. This is particularly important when the legacy system is running on-premises. This allows us to isolate and control the number of cloud accounts that have access to the on-premises network. It also makes it possible to simply shut down a cloud account once a legacy system is retired.

Your open API is another candidate to isolate and treat as an autonomous subsystem in a dedicated cloud account. This subsystem will integrate with the core subsystems, like any other subsystem, but it will expose your external domain model to the outside world, which you will have little to no control over. It will also be responsible to a single generic external actor (which you define) that represents the prototypical external system your open API is designed to support. Therefore, it is best to create this outer layer as a bulkhead to isolate security concerns and support the strongest backward compatibility guarantee.

Now let’s dive into various applications of the ESG pattern.

Integrating with third-party systems

One easy way to accelerate the pace of innovation is to avoid building software that you don't need to build. If a piece of functionality is not part of your core competency's value proposition, then you should not build it. More than likely, it is already offered as a service by a third party.

This is akin to the serverless-first approach. We leverage a fully managed, third-party capability and defer commitment so that we can focus on building our value proposition and create knowledge about what works and what does not work. Then, once we know what capabilities are actually required, we can make an informed decision to stay with the third party, switch to another provider, or implement the functionality ourselves.

To facilitate this process, we employ the ESG pattern to decouple this decision from the core of the system. We wrap the third-party system with an ESG service that adapts the external system to the core model. Then we can simply substitute with a different ESG service when we change directions. We can even run multiple implementations simultaneously for different scenarios.

Let's look at the egress flow for performing an action and the ingress flow for receiving updates about the progress of the action.

Egress – API call

Software as a Service (**SaaS**) products typically provide a well-documented open API for invoking their functionality. From our food delivery example, this includes the likes of authorizing a payment or sending email messages and push notifications to customers.

These egress flows react to an internal domain event and invoke an action on a third-party system to perform a specific business function. *Figure 7.4* depicts the resources involved:

Figure 7.4: Egress – API call

The domain events of interest are available on a `stream` in the event hub. The `listener` function is a stream processor that consumes from the stream and filters for the right event type, such as `OrderSubmitted`. It transforms the data to create the external request and invokes the action via a `REST` `POST`**:**

```
...
    .map(toPostRequest)
    .ratelimit(opts.number, opts.ms)
    .through(post())
...
```

One thing of note is that third-party APIs are typically throttled, and pricing may be based on the rate at which they are invoked. Therefore, it is recommended to add explicit backpressure with a rate-limiting feature. We covered this and other stream processing optimization techniques in *Chapter 4, Trusting Facts and Eventual Consistency*.

In *Chapter 4, Trusting Facts and Eventual Consistency*, we also covered the important role that atomic actions play in resilient systems. In this context, it means that we should not attempt to invoke an external action and then produce an internal event about the success of the action in the same unit of work. Instead, we will rely on the third-party system's webhook feature to notify us about the success of the action.

Ingress – webhook

SaaS products provide webhooks as a generic way to integrate with other systems. These allow other systems to register a callback so that the SaaS product can notify an external system about internal changes and progress updates. Some examples include a payment service emitting an event when a payment is completed or an email service emitting an event when a message is flagged as junk.

These ingress flows integrate with a webhook to receive external domain events and transform them into internal domain events so that the system can react as needed. *Figure 7.5* depicts the resources involved:

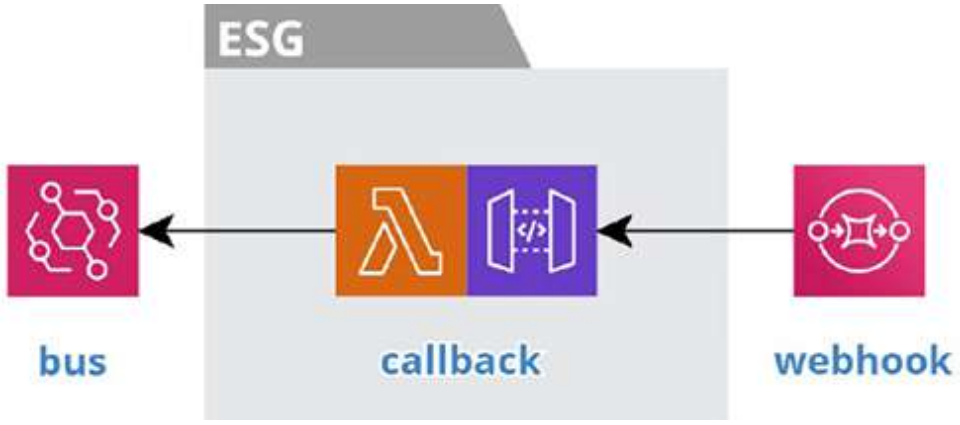


Figure 7.5: Ingress – webhook

The third-party system emits events via its webhook feature. The ESG uses an *API gateway* to implement a **callback** endpoint that will receive external events from the **webhook**. The **callback** function transforms the external format to the internal event format and forwards the internal domain event to the event hub, so that core BFF and Control services can react to the event.

Asynchronous request response

The interactions with external systems are typically the least resilient part of a system because they require a synchronous call to another system that you do not control. This is why it is preferable to have ESG services that are completely asynchronous and react to events from other services. This supports the majority of scenarios where a fire-and-forget flow is appropriate, such as when one user performs a step and another user performs the next step after any automated steps have been successfully completed in between.

However, there are scenarios where a single user performs a step and needs a response from an external system before continuing to the next step. It is important that the user proceeds to the next step as quickly as possible, but the user shouldn’t have to wait for a flaky and/or slow response. As an example, in our food delivery system, it can take some time for a restaurant to acknowledge receipt of a customer’s order. In these scenarios, we can employ an asynchronous request-response flow to make the user experience more resilient and responsive. *Figure 7.6* depicts the resources involved:

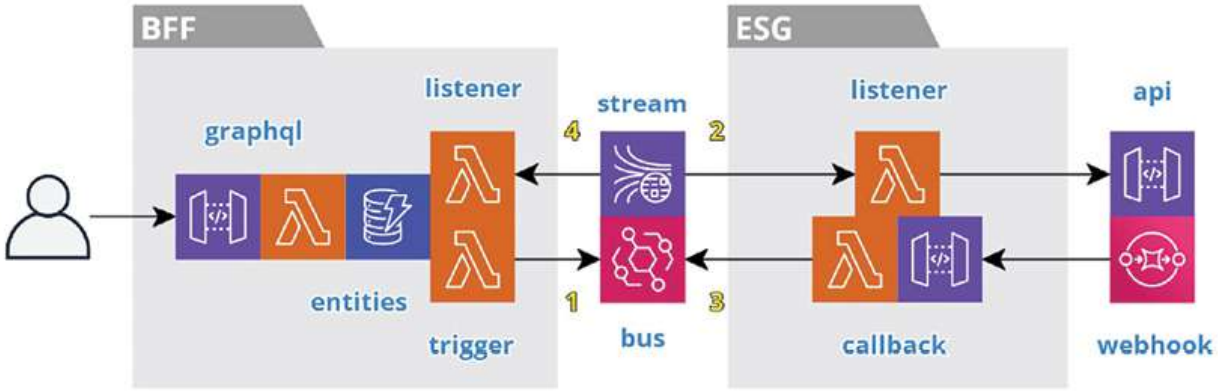


Figure 7.6: Async request response

The BFF's `graphql` function provides a mutation to initiate the request. It stores a request object in the `entities` datastore. The `trigger` function publishes a request event, such as `<entity>-<action>-requested`.

The ESG's `listener` function consumes the `request` event and makes the external API call. The external system's `webhook` feature invokes the `callback` with the results of the action. The `callback` function publishes a response event, such as <`entity>-<action>-completed`.

The BFF's `listener` function consumes the response event and updates the `request` object in the `entities` store. Meanwhile, the frontend polls the BFF's `graphql` function to query for the status of the request. The polling implements an exponential backoff and results in an error after several attempts. If the response arrives later, then a fresh query will return the results, which provides for more resilience. Alternatively, a live update approach can be used, such as was presented in *Chapter 3*, *Taming the Presentation Tier*.

Synchronous ESG/BFF hybrid

Sometimes, the asynchronous request-response approach is just not suitable for a given use case. For example, using the asynchronous model to validate an address against a postal service's address database would not make for a smooth user experience. In this case, we can follow a synchronous ESG/BFF hybrid approach. *Figure 7.7* depicts the resources involved:

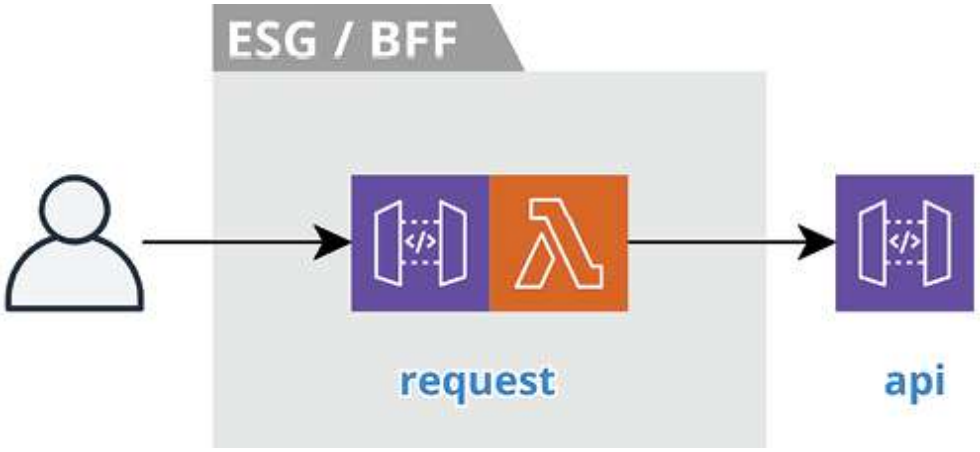


Figure 7.7: Synchronous ESG/BFF hybrid

In this scenario, the ESG exposes an API gateway endpoint just like a BFF service. The function translates the internal request format to the external format and makes a synchronous request to the external system. Then it transforms the external response to the internal format and returns the response.

Of course, when we make a compromise like this and tightly couple one system to another, we must prepare for the consequences. In this case, we need to perform more error handling and decide what to do when the external system is not available. Plus, we should expose an internal request/response format to the frontend to minimize coupling so we can easily substitute a different external system as circumstances change.

Third-party systems are the most obvious kind of external system. However, as a system grows larger, we will need to divide it into multiple related

subsystems that need to talk to each other. Next, we will look at how to integrate these subsystems into a unified whole.

Integrating with other subsystems

For any large system, it is necessary to divide the system into a set of manageable subsystems. In *Chapter 2, Defining Boundaries and Letting Go*, we discussed the concept of autonomous subsystems. Like autonomous services, these subsystems employ *bulkheads* to protect themselves from each other. We fortify each autonomous subsystem in its own cloud account, each has its own event hub, and we only expose a set of *external domain events* for inter-subsystem communication.

We can then create arbitrarily complex systems by connecting autonomous subsystems in a simple fractal pattern. This system topology is depicted in *Figure 7.8*:

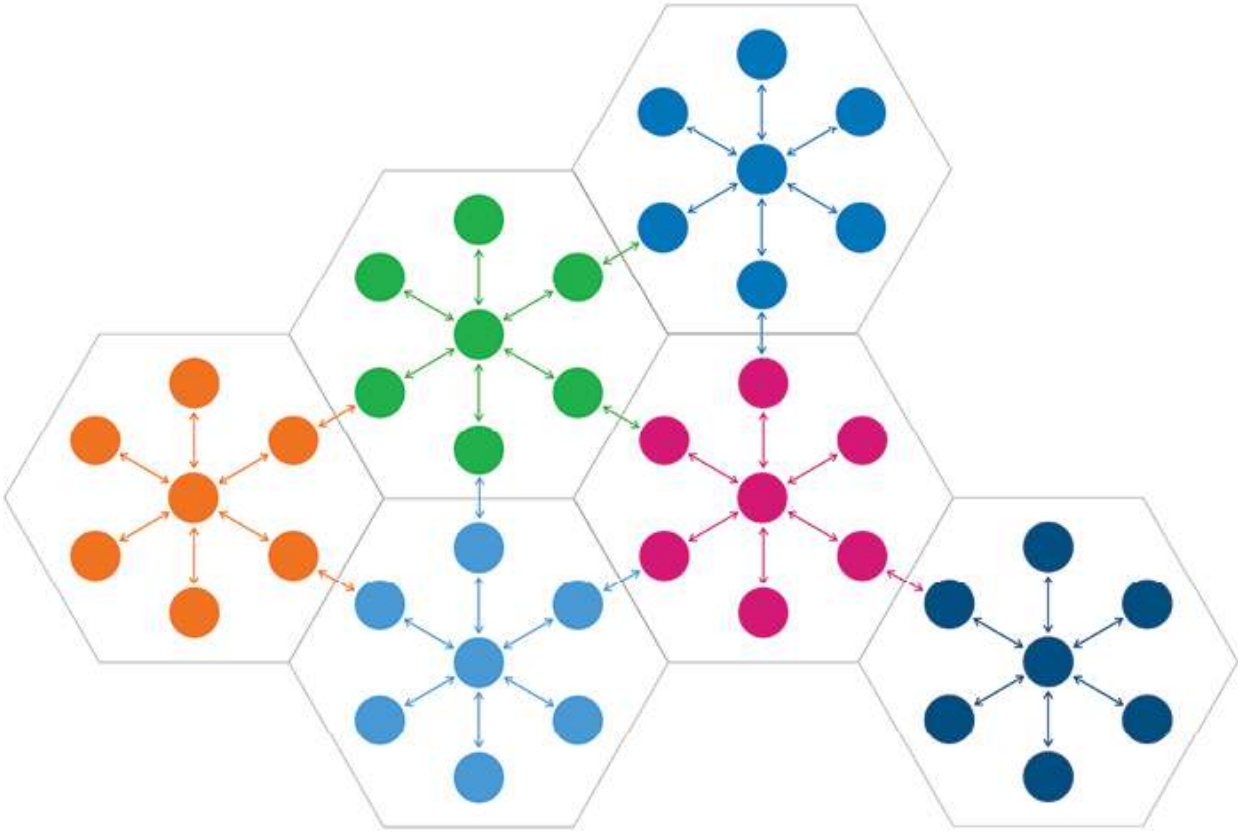


Figure 7.8: Autonomous subsystem topology

The nodes in the diagram that are connected across the subsystems represent the *egress* and *ingress* ESG services. In this section, we will cover the tactical aspects of transforming between internal and external domain events and connecting the subsystems across accounts.

Egress – upstream subsystem

Upstream subsystems make a set of external domain events available and route them to downstream subsystems. *Figure 7.9* depicts the inter-subsystem egress flow and the resources involved:

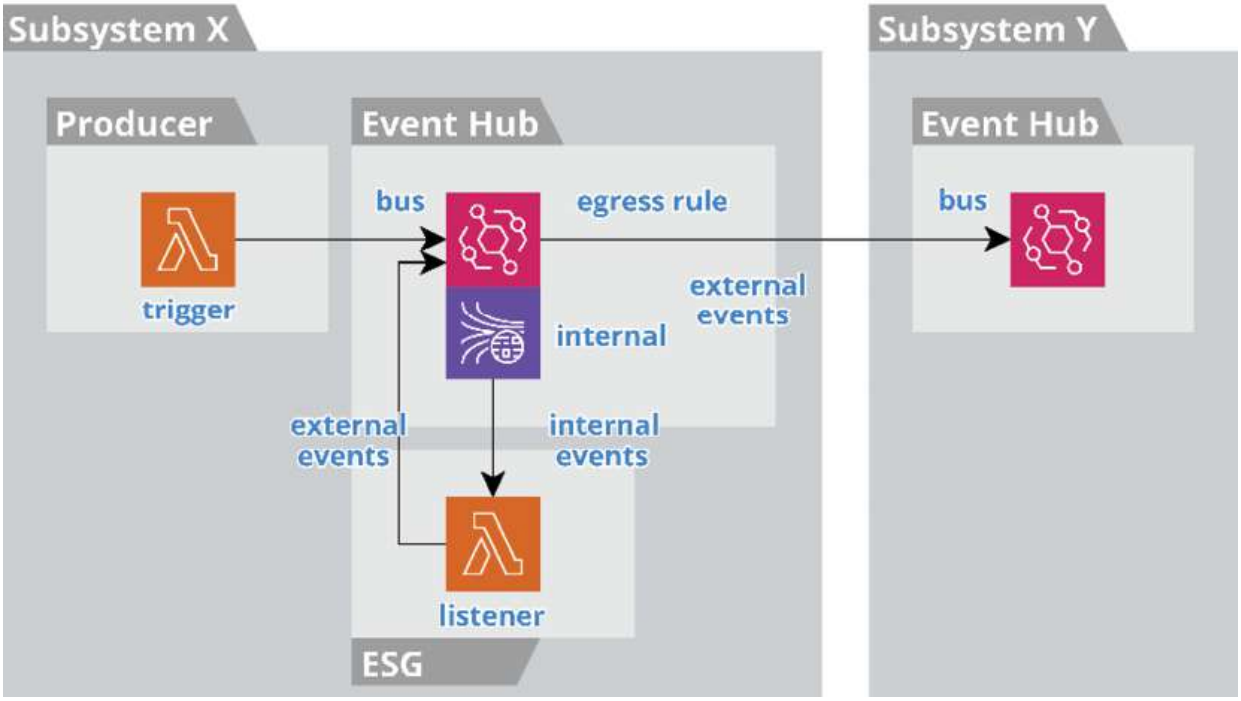


Figure 7.9: Inter-subsystem egress flow

The autonomous services within a subsystem exchange **internal** domain events. We only guarantee the backward compatibility of these internal event types within the subsystem. The **Producer** services publish internal domain events to the subsystem's **bus** as usual and remain completely decoupled from downstream subsystems.

Each subsystem produces a set of external domain events that contain the information it is willing to share and support. This hides the internal dirty laundry of a subsystem and provides stronger backward compatibility guarantees to downstream subsystems. One or more *egress* **ESG** services consume the internal domain events and *transform* them into **external** domain events. Then the ESGs publish these events to the **bus** as well.

From here, the event bus service, such as AWS EventBridge, forwards events from the **bus** in one account to the **bus** in another account. We define

one or more **egress rules** to relay the external domain events to downstream event hubs.

Here is an example of an egress rule for AWS EventBridge that routes all external domain events to one or more downstream subsystems. The ESG services set the `source` field to `external` to simplify the routing rules. We add a target for each downstream `bus` that will receive the upstream subsystem's events:

```
EgressEventRule:
  Type: AWS::Events::Rule
Properties:
    EventPattern:
      source: ['external']
    Targets:
      - Id: SubsystemY
Arn: arn:aws:events:*:123456789012:event-bus/event-hub-bus
    ...
```

Both subsystems must agree to the exchange of events. We establish a trust agreement by configuring permission policies in both accounts stating which accounts a subsystem is willing to send events to and receive events from.

Now let's complete the picture by looking at a downstream subsystem's ingress flow.

Ingress – downstream subsystem

At this point, upstream subsystems have made their external domain events available and routed them to downstream subsystems. Next, each downstream subsystem makes the events of interest available to its own

services. *Figure 7.10* depicts the inter-subsystem ingress flow and the resources involved:

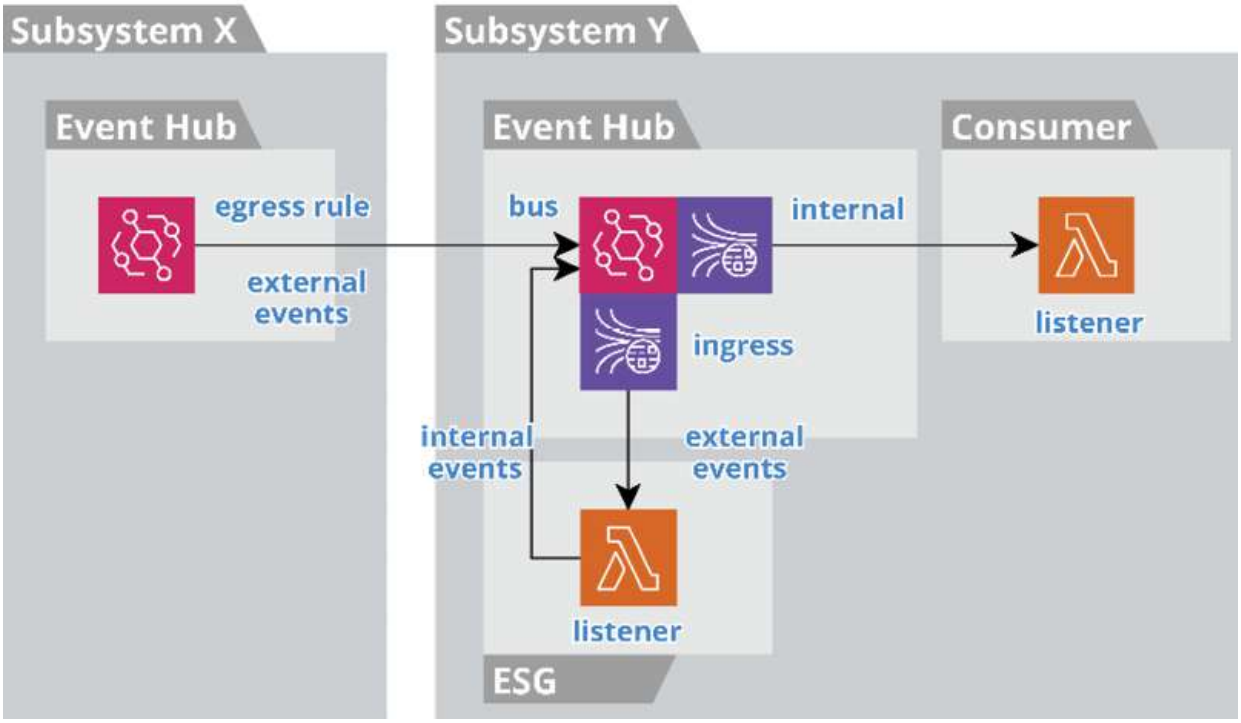


Figure 7.10: Inter-subsystem ingress flow

The **bus** of a downstream event hub receives **external** domain events from an upstream subsystem and routes the events to an **ingress** channel to isolate them from internal domain event flows. One or more *ingress* **ESG** services consume the external domain events and transform them into **internal** domain events. Then, the ESGs publish the internal domain events to the **bus** where we route them to the **internal** channels for further processing, as usual, by **Consumer** services.

Note that the **ingress** channel is optional. If the volume is low, then it may not be necessary. You can start without it and add it as needed. It can also be excluded in lower environments to control costs. We covered channel topology in *Chapter 4*, *Trusting Facts and Eventual Consistency*.

This variation of the ESG pattern allows subsystems to evolve independently. We only need to modify the ESG logic when the definition of an internal or external domain event changes. We follow the robustness principle when a change is necessary so that both sides do not have to change at the exact same time. This anti-corruption layer is the primary benefit of this ESG pattern scenario. We will cover the details of applying the robustness principle in *Chapter 11*, *Choreographing Deployment and Delivery*.

Implementing subsystems on the same cloud provider allows us to leverage the features of common services. Next, we will look at integrating subsystems that we deploy on different cloud providers.

Integrating across cloud providers

Polycloud is the notion that we should use the right cloud provider for the job on a service-by-service basis. This is similar to the concepts of *polyglot programming* and *polyglot persistence*. We will discuss the business value supporting this approach in *Chapter 13*, *Don't Delay, Start Experimenting*. In this section, we will cover the tactical aspect of connecting subsystems that run across cloud providers.

Fundamentally, polycloud is employed at the service level but, from a practical standpoint, it operates at the subsystem level. For example, if you only have a single service that you want to run on a different cloud provider, then we can think of it as a subsystem consisting of a single service. However, there would most likely be several related services that all benefit

from running on the alternate cloud provider and others that are just better off running as closely as possible to related services.

This subsystem point of view makes integrating across cloud providers much easier because it builds on the inter-subsystem integration approach that we use between all autonomous subsystems. The subsystem ingress flow we just covered remains completely unchanged, while the subsystem egress flow just needs a small tweak. An example will make this clear.

In this example, subsystem **X** runs on cloud provider **X** and subsystem **Y** runs on cloud provider **Y**. Each of these subsystems benefits from the *bulkheads* of not only separate cloud accounts but also separate cloud providers. We will follow the same bulkhead approach of exchanging external domain events between these subsystems:

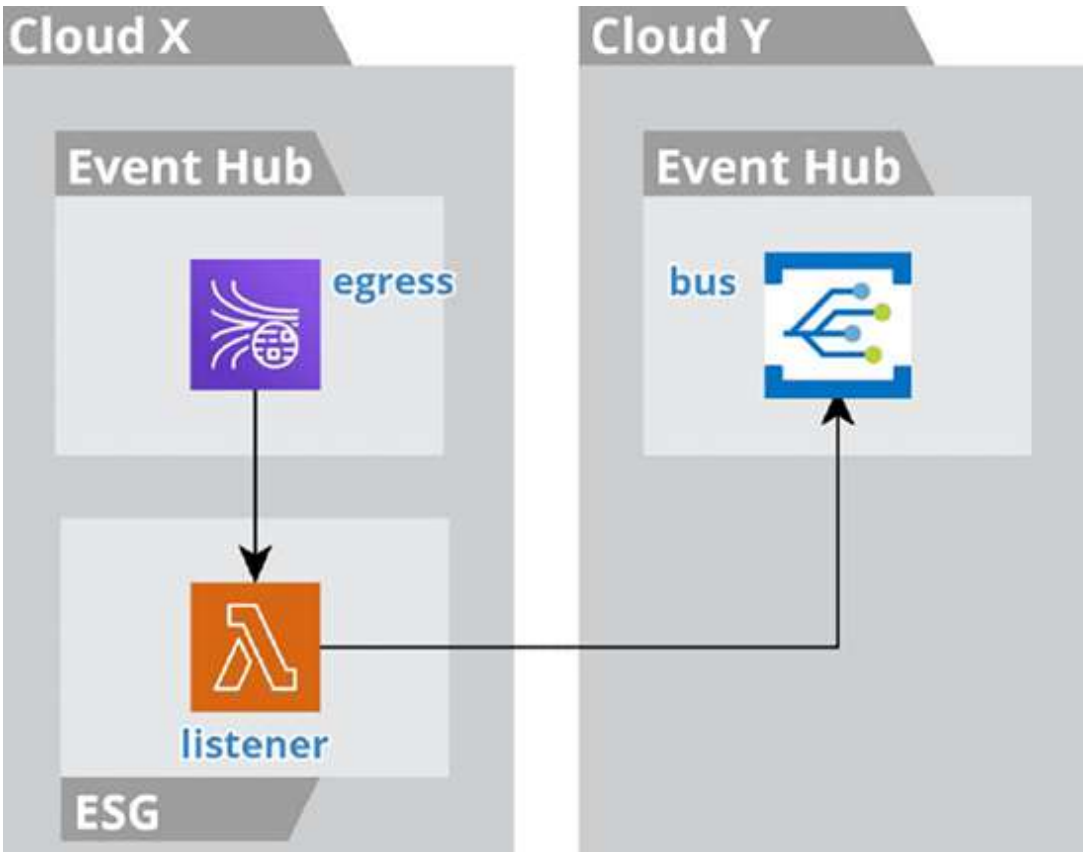


Figure 7.11: Polycloud egress

As we previously covered, subsystem **X** will have egress ESGs that handle transforming internal domain events into external domain events. From here, we cannot rely on out-of-the-box cloud service features to route events between cloud providers. For example, if subsystem **X** is using AWS EventBridge as its event bus and subsystem **Y** is using Azure Event Grid or GCP Pub/Sub, then we need to implement a forwarding mechanism.

To accomplish this, we simply add an additional ESG to bridge the gap. The **listener** function of this ESG consumes external domain events from an **egress** channel and sends them to subsystem **Y**. It batches up events and invokes the API provided by the bus of the downstream cloud provider. The *ingress flow* in the target subsystem remains the same with ESGs dedicated to transforming the external domain events into internal domain events.

Integrating subsystems and third-party systems is challenging because the systems have different owners. Legacy systems present additional challenges because they are often not designed to support integration. Next, we will look at techniques to overcome these hurdles.

Integrating with legacy systems

Delivering new business value at a faster pace is the major goal of digital transformation. Achieving this goal usually involves migrating from the existing (that is, legacy) systems. This migration process does not happen overnight. Therefore, it is crucial that we do not put the existing systems at risk. They need to continue to deliver business value until they are retired.

To provide this continuity of business, we follow an event-first migration approach that iteratively adds new business value and incrementally upgrades existing features, until we no longer need the legacy systems. We refer to this approach as the **Strangler pattern**. We will discuss the business value supporting this approach in *Chapter 13*, *Don't Delay, Start Experimenting*. In this section, we will cover the tactical aspects of integrating with a legacy system.

There are two main objectives for this approach:

- We do not want to modify the existing systems, if possible. This is consistent with the **Open-Closed principle**. Any change puts an existing system at risk, so we will employ the least invasive alternatives.
- We want to simply turn the existing systems off when we no longer need them, with zero downtime and zero modifications to the new system. This is consistent with the **LSP**. In essence, we want to choreograph a smooth evolution from the old to the new.

We effectively need to turn legacy systems into autonomous services by wrapping them with the ability to produce and consume domain events. A legacy system will exchange events with the new system so that both can react to each other, maintain their own state (that is, data), and stay synchronized. This enables users to use the old and the new concurrently. We can incrementally transition features from old to new and easily fall back to the old when we miss the mark.

At its core, this is the time-honored approach of **Enterprise Application Integration** (**EAI**) that we discussed in *Chapter 1*, *Architecting for Innovation*. This approach is the heart of our evolutionary architecture, but it is most evident when integrating with legacy systems.

In this section, we will enumerate a set of common legacy integration scenarios. The number of potential scenarios may seem limitless, bounded only by the number of legacy systems. But in all cases, we are integrating the legacy system into the event hub so that it can act as another spoke in the overall system.

Ingress – Change Data Capture

Many legacy systems do not have a mechanism that emits events to notify the outside world when their state has changed. Systems that do have such a mechanism may not emit the right level of detail. Fortunately, there is a non-invasive way to add this capability to many systems.

Change Data Capture (**CDC**) is an approach that leverages a database's *transaction log* to identify when data has changed so that the changes can be captured and shared with other systems. CDC tools are non-invasive because they only need the ability to read the transaction log. Every insert, update, and delete operation is recorded in the transaction log.

A database can use a transaction log to completely replicate its state. This means that CDC can provide all the details needed to synchronize the legacy system to the new system. *Figure 7.12* depicts the resources involved:

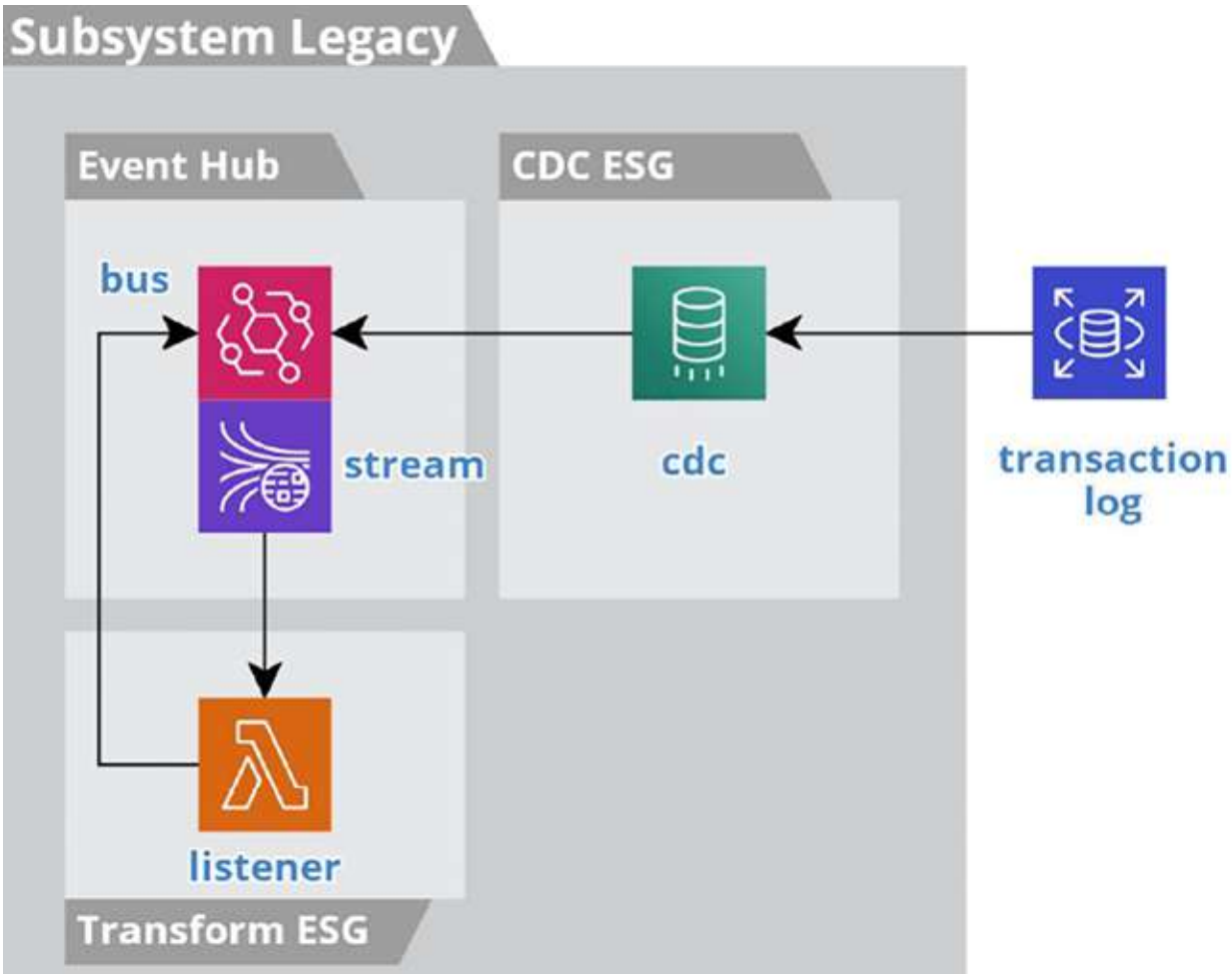


Figure 7.12: Ingress – CDC

First, we are isolating access to the legacy database within its own autonomous subsystem and cloud account. Next, a CDC tool, such as AWS's fully managed **Database Migration Service** (**DMS**), is granted access to the legacy database's transaction log. The CDC tool publishes an event to the event hub for every insert, update, and delete operation performed on all tables of interest. These internal events contain the raw details about the individual statements that changed the data.

One or more *ingress* ESG services transform the database events into the external domain events that the new system will consume. To create the external domain events, an ESG will likely need to aggregate the contents of

several low-level database events. We will cover this enrichment process in the *Tackling common data challenges* section later in this chapter. From here, we route the external domain events to downstream subsystems following the approach discussed previously for integrating autonomous subsystems.

You can find a CDC example here: https://github.com/jgilbert01/templates/tree/master/template-dms-ingress-gateway.

Egress – Direct SQL

Many legacy systems do not have an external interface for invoking business logic. In these systems, the logic to update the database is usually coupled with the presentation logic. They often rely on **Extract, Transform, and Load** (**ETL**) processes for periodically ingesting external data. In this scenario, we can use Direct SQL to create a real-time integration. *Figure 7.13* depicts the resources involved:

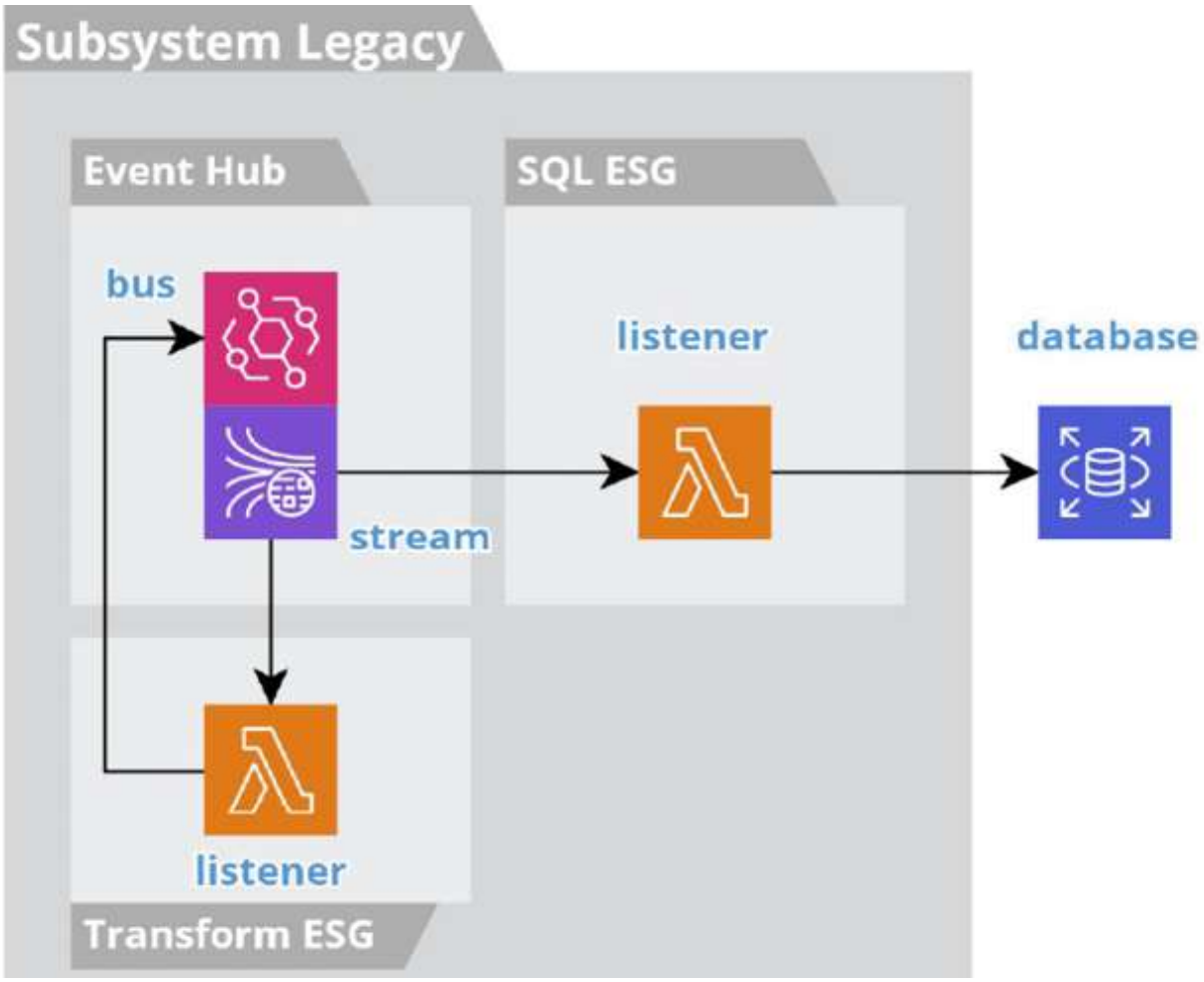


Figure 7.13: Egress – Direct SQL

Again, we are isolating access to the legacy database within its own autonomous subsystem and cloud account. This egress flow would be part of the same subsystem that contains the CDC ingress flow and would share the same access to the legacy database. External domain events from upstream subsystems are routed to this subsystem following the approach discussed previously for integrating autonomous subsystems.

One or more *egress* ESG services transform the external domain events into internal domain events that contain one or more SQL statements. These events form an audit trail in the event lake of the changes made to the legacy database. The SQL ESG executes the SQL statements in a transaction

against the legacy database. Note that SQL statements that generate sequence numbers are not idempotent. In this case, you may need to incorporate some of the techniques covered in the *Tackling common data challenges* section.

Egress – circuit breaker

Autonomous services are designed to avoid the need for the Circuit Breaker pattern. This is accomplished by eliminating all inter-service synchronous communication, leveraging fully managed, highly available, serverless resources, and creating inbound and outbound bulkheads. We covered this in detail in *Chapter 4, Trusting Facts and Eventual Consistency*, and *Chapter 5, Turning the Cloud into the Database*.

Unfortunately, the Circuit Breaker pattern may be unavoidable when integrating with external systems. Egress flows typically require synchronous calls to the external system and a circuit breaker may be necessary if the external system is not reliable. In this case, the ESG encapsulates these details and accommodates the impedance mismatch.

A third-party SaaS system is hopefully reliable enough and you may only need backpressure and rate-limiting to avoid throttling. On the other hand, legacy systems are often less reliable and take longer to recover when they experience a failure. In this case, we need to ensure that an ESG does not exacerbate the problem by continuing to overload the external system. We also want to avoid the increase in cost that results from the stream processor function repeatedly waiting for the calls to timeout and retry.

The solution is to include a circuit breaker with the ESG. The state machine diagram in *Figure 7.14* depicts the approach. The circuit breaker is in the **closed** state when the external system is operating properly, and the external

calls are mostly successful. When the calls reach the **failure threshold**, then the circuit breaker transitions to the **open** state. The ESG will stop calling the external system for a period of time to give the external system a chance to recover. After the **timeout** period, the circuit breaker transitions to the **half-open** state. In this state, the ESG will try the next call to test whether the external system has recovered. If the call is successful, then the circuit breaker transitions to the **closed** state, and if it is a failure, then it transitions back to the **open** state and waits again before attempting another test:

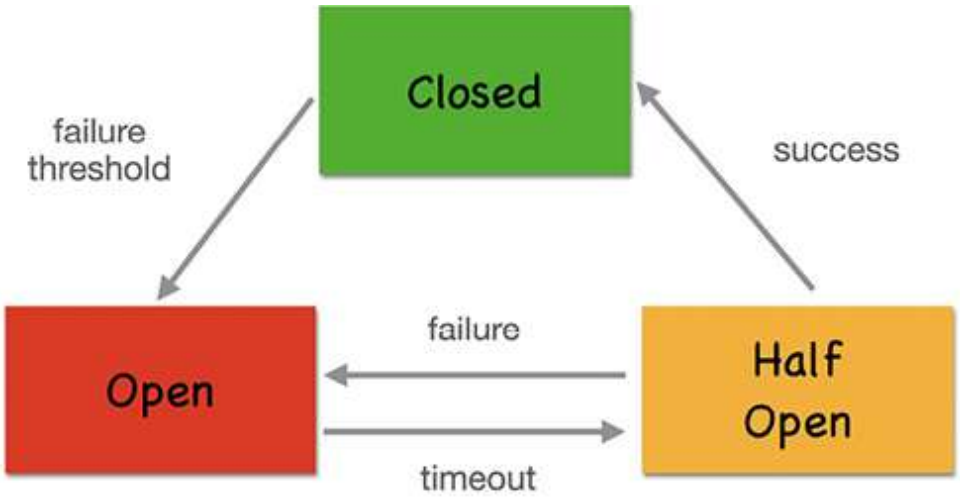


Figure 7.14: Circuit breaker state machine

There are many ways to implement the circuit-breaker pattern. For our asynchronous listener functions, we can implement the pattern by disabling and re-enabling the function. When disabled, the function will not process events and they will accumulate in the channel. When we re-enable the function, it will pick up where it left off. In between, we are giving the external system time to recover. *Figure 7.15* depicts the resources involved:

Figure 7.15: Egress – circuit breaker

We can use an AWS CloudWatch alarm to open and close the circuit. For example, the alarm can watch the error count for the listener function and compare it to the invocation count. When everything is running smoothly, then the alarm is in the OK state (that is, Closed) and events flow normally to the external system. When the error count is too high, then the alarm will transition to the ALARM state (that is, Open). At this point, we can have a utility function disable the event source mapping (that is, subscription) between the stream and the listener function.

After the evaluation period expires, the alarm will transition to the INSUFFICIENT_DATA state (that is, Half Open) because there are no invocations or errors now that we have disabled the listener function. At this point, we can have a utility function re-enable the event source mapping and set the batch size to one so that we can test the external system with a small number of events. If the external system is still unavailable, then the alarm will transition back to the ALARM state and disable the listener again; otherwise, it will transition to the OK state and the utility function will increase the batch size to the original value. Keep in mind that a given

channel can only store events for a limited period. If an outage lasts longer, then we will need to replay the expired events from the event lake.

Ingress – relay

Some legacy systems do have a mechanism that emits events or messages to notify the outside world when its state has changed. For example, a Java-based system may have one or more **Java Messaging Service** (**JMS**) topics or queues that it uses to publish events or send messages. In this scenario, we can plug into these channels to listen for events and receive messages. We can create an ESG service that will relay the events from the topic on-premises to the bus in the cloud. *Figure 7.16* depicts the resources involved:

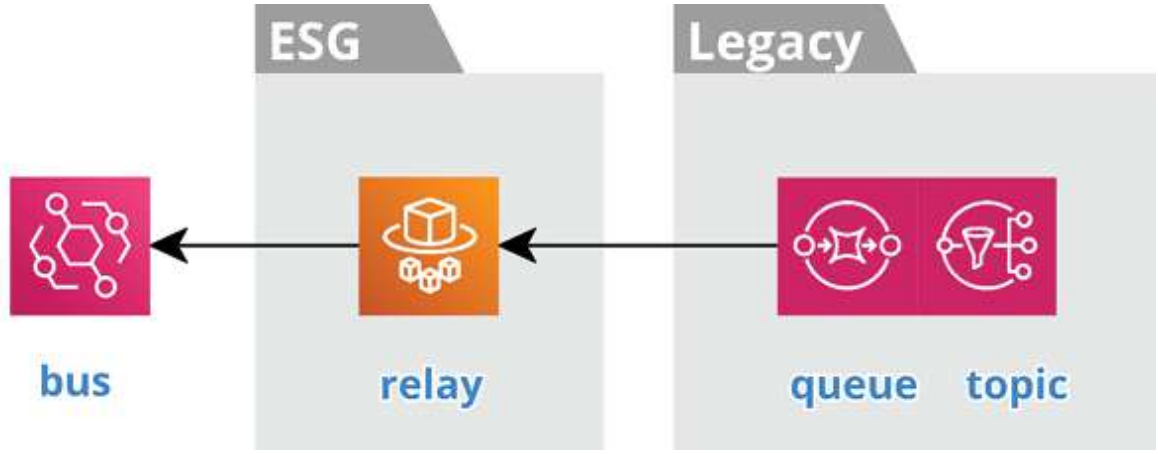


Figure 7.16: Ingress – relay

First, we need to create a queue on-premises that the relay will use to receive messages from the JMS topic. We will need a long-running process to poll the queue for messages, so we will use a serverless container service, such as AWS Fargate. The relay should do as little transformation as possible and focus on the technical challenges of connecting to the on-premises environment.

The relay will pull messages from the queue, normalize them by wrapping them in our standard event envelope and publish the external events to the bus. The events will contain the raw external messages. This will create an audit trail in the event lake of exactly what we received from the external system. Another ESG downstream will consume the raw external events and transform them into internal domain events for use within the subsystem.

We have just covered a variety of scenarios for integrating with legacy systems, but it is far from an exhaustive list. Another approach that can be used to integrate with legacy systems, sister systems, and external systems in particular is the creation of an open API. We will cover these scenarios next.

Providing an open API and SPI

Non-invasive integration is one of the goals of the ESG pattern. This is important when you do not have control of the other systems, but it means that you are taking on the responsibility of implementing the integrations that adapt the other systems to your system. This is only manageable for a limited number of systems.

If you are implementing your own SaaS product, then the number of external systems that want to integrate with your system can be limitless. In this case, the only manageable solution is to provide your own *open interface* and make the external systems responsible for integrating with your system.

Open interfaces expose your external domain model to the outside world. They act as a façade to hide the inner workings and provide strong backward compatibility guarantees. They simplify the process of integrating with your system and enable external systems to take on the responsibility for

connecting to your system and *transforming* between their external model and your external model.

An open interface includes an **Application Programming Interface** (**API**), so that other systems can call your system, and a **Service Provider Interface** (**SPI**) so that your system can notify other systems about important events. They support asynchronous ingress and egress flows to receive and emit events and synchronous flows to perform actions and queries. You may also provide a self-service developer portal so that organizations can register to use the open interface.

There are two types of ingress APIs that we will cover in this section. Most SaaS systems only provide a more traditional synchronous ingress API, but we will also cover an asynchronous ingress API that supports an event-first approach. We will then look at two types of egress interfaces: a webhook to push events to external systems and queries allowing external systems to pull desired data.

Ingress API – event

This *ingress* interface allows external systems to send events to your system. It is essentially the inverse of an egress webhook. It allows external systems to notify your system about their state changes. In other words, they have already performed an action and they want to give your system an opportunity to react. Following our food delivery example, when a restaurant updates its menu, then it will need to notify all its food delivery partners. *Figure 7.17* depicts the resources involved:

Figure 7.17: Open API – ingress – event

The ESG provides a RESTful endpoint to **receive** events from external systems. The interface defines an explicit set of external domain events that it accepts. To improve throughput, the interface may support receiving events in batches. Each external system implements its own egress ESG service that will transform its domain events to your external domain events and **relay** them to your endpoint.

The **receive** function validates the incoming events and forwards them to the event hub's **bus**. From here the ESG variation we covered in the *Integrating with other subsystems* section will forward the events downstream to elicit reactions from other subsystems.

Ingress API – command

This *ingress* interface is the flip side of the third-party egress integration. It allows external systems to submit a request for your system to perform an action on its behalf. Following our food delivery example, this interface may

allow another system to request a driver. *Figure 7.18* depicts the resources involved:

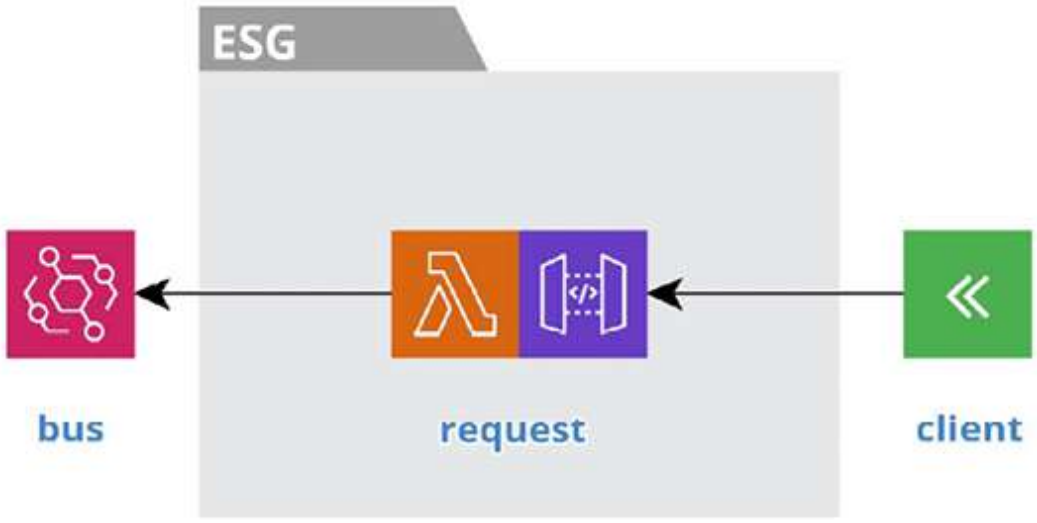


Figure 7.18: Open API – ingress – command

The ESG provides a RESTful endpoint so that an external system can submit a **request** for a specific command to perform an action. Each external system implements its own egress ESG service that will transform its domain entities to your external domain entities and act as a **client** to call your endpoint.

The actual details regarding how to perform the requested command are not the responsibility of this ingress ESG. The **request** function validates the incoming request and publishes an event that the request was submitted. From here, the ESG variation we covered in the *Integrating with other subsystems* section will forward the event to a downstream subsystem so that a downstream service can perform the actual work.

The act of receiving the request and producing the event is an atomic transaction with a low likelihood of failure and the event is an audit trail of receiving the request. The external system can trust that the request will be

fulfilled, and it can receive status events through the webhook ESG, which we will cover next.

Egress SPI – webhook

This *egress* interface allows external systems to register for notifications about state changes in your system. In other words, your system has already performed an action and the external system wants the opportunity to react. Following our food delivery example, when an order has been submitted, the restaurant needs to be notified so that it can start preparing the food. *Figure 7.19* depicts the resources involved:

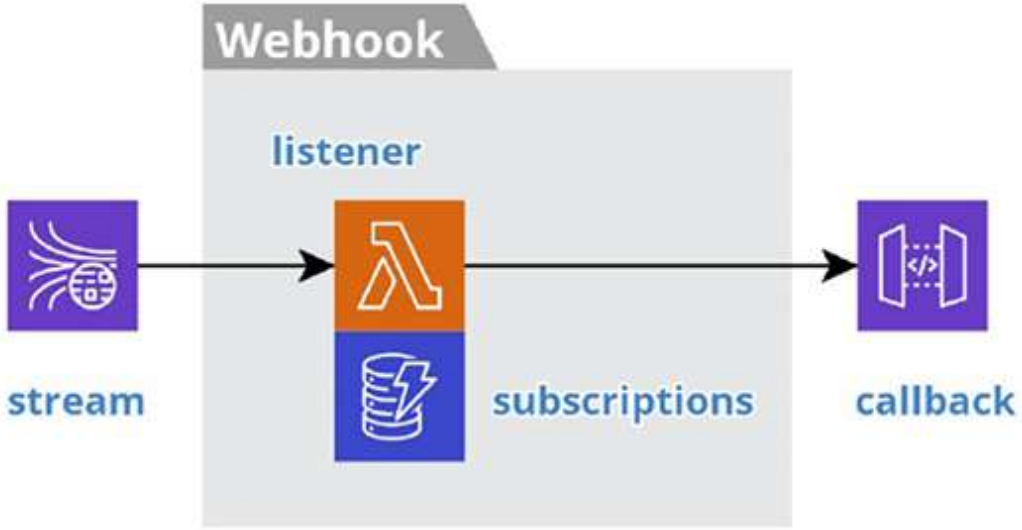


Figure 7.19: Open SPI – egress – webhook

The external system provides a RESTful endpoint that implements a **callback** to receive your external domain events, as we covered in the *Integrating with third-party systems* section. A developer will register a **callback** endpoint using your developer's portal, which would be implemented using the BFF pattern, as we covered in *Chapter 6, A Best Friend for the Frontend*.

The BFF will emit events for these registrations. The ESG's **listener** function will consume these registration events and cache the **subscriptions** in a datastore.

The **listener** function will also consume external domain events and look up registered subscriptions. For each subscription, the **listener** will invoke the registered **callback** endpoint with the external domain event. For example, restaurants would subscribe to **OrderSubmitted** events. When an **OrderSubmitted** event is received, the **listener** looks up **subscriptions** for the restaurant specified in the order and passes the event to the callback URL registered for the **OrderSubmitted** event.

Egress API – query

This *egress* interface allows external systems to pull data from your system on demand. Following our food delivery example, third-party systems might want read access to the restaurants and their menus. *Figure 7.20* depicts the resources involved:

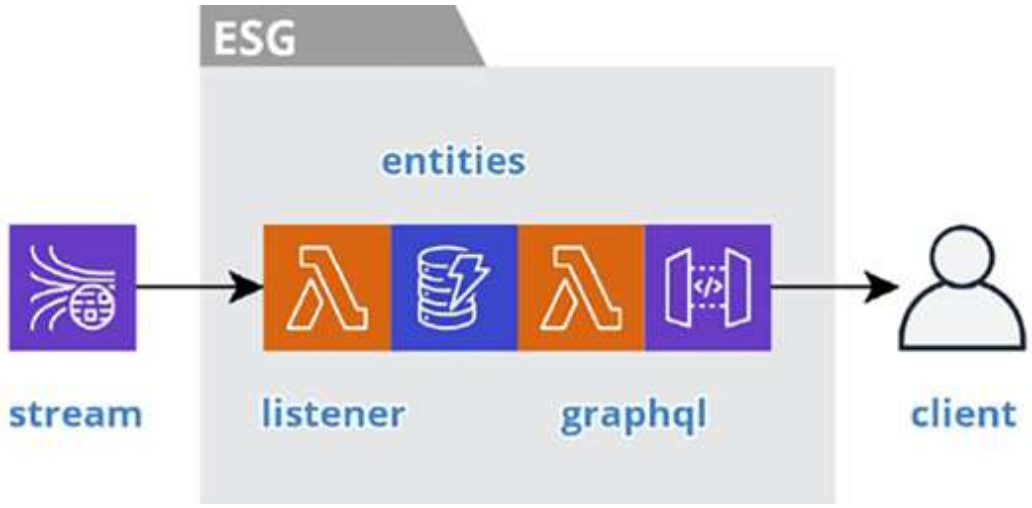


Figure 7.20: Open API – egress – query

The **listener** function consumes external domain events and caches the data in the **entities** table. A **graphql** or RESTful interface is provided to allow external clients to query data. In this case, **graphql** is preferred because its client-driven nature will make it easier to provide backward compatibility to a large set of external systems, as we discussed in *Chapter 6, A Best Friend for the Frontend*.

One thing you may have noticed is that this API looks very much like a BFF service. In essence, it is a BFF that is designed to support many external frontends. It employs the CQRS pattern to cache data, specifically allocates capacity for external systems, and creates a bulkhead for the rest of the system. If the interface needs to support mutations, then it would look like a full BFF with a **trigger** function as well.

But we make a distinction here because it breaks the BFF pattern of the frontend and backend being owned by the same team. This distinction is important because of the impact it has on the pace of innovation. This open interface is intended to change slowly and provide a strong guarantee of backward compatibility. It is included here for completeness, but it is recommended that external systems use webhooks to create their own materialized views instead.

Up to this point, we have covered a variety of scenarios for integrating with external systems. Next, we change gears a little and cover some cross-cutting concerns related to common data challenges.

Tackling common data challenges

Let’s look at some common considerations that are applicable across many of the ESG scenarios we covered throughout this chapter, such as idempotence, data enrichment, latching, and resynchronizing slow data.

Idempotence

Idempotence is an important piece of the event-first approach. We covered different approaches in *Chapter 4*, *Trusting Facts and Eventual Consistency*, and *Chapter 5*, *Turning the Cloud into the Database*. The external systems that ESGs integrate with may or may not support idempotence. For example, a third-party SaaS product most likely will support it, but a legacy system probably will not. The legacy system’s API will dictate which approach we can use, such as using Direct SQL to implement an inverse OpLock.

If the external system does not provide idempotence, then we can implement it in the ESG with a micro event store. *Figure 7.21* depicts the resources involved:

Figure 7.21: Egress – idempotence

For an *egress* flow, the **listener** function inserts an event in the **entities** store. The first time an event is stored, the **trigger** function will be invoked and perform the logic needed to integrate with the external system. If the **listener**

receives the same event more than once, the **trigger** will only be invoked once.

It may be necessary to provide idempotence on the ingress flow of your external interface as well. *Figure 7.22* depicts the resources involved:

Figure 7.22: Ingress – idempotence

For an *ingress* flow, the **receive** function stores the incoming events in the **events** store and the **trigger** function is invoked for each unique event. To support idempotence, the external system must provide a deterministic event ID; otherwise, an event that is logically a duplicate will be treated as unique when the ID changes.

Enriching data

An individual event may not contain all the information needed to produce an external domain event or make a call to an external system. For example, the ingress CDC flow produces internal events at the granularity of individual table rows that may only contain foreign keys to related entities. In this case, the ESG needs to enrich the data before proceeding. The ESG

can accumulate the related data by consuming more events and caching the needed information. *Figure 7.23* depicts the resources involved:

Figure 7.23: Data enrichment

In this example, a specific ESG is responsible for producing external domain events for one or more aggregate domains. The **listener** function consumes events from all the tables that make up the aggregate domain and accumulates the data in an **entities** table following the *single table pattern*, which we covered in *Chapter 5, Turning the Cloud into the Database*.

The **trigger** function is invoked when data is added. It retrieves all the related data and will produce an external domain event once all the needed data is present.

For example, in a typical relational model, for something such as a purchase order, there will be reference data for products and the customer, a master record for the order that references the customer, and then a record for each line item that references the master order record and a product record. The reference data will usually arrive first (possibly long before), and then the master record arrives, followed by each line item. An **OrderItemAdded** external domain event can be produced for each line item as it arrives, so long as the master record and reference data are available. If the events arrive out of order, such as if the master record arrives after some line items,

then we can produce the events for those line items once the master record arrives.

Latching and cross-referencing

Latching is an approach for controlling infinite loops in a situation where we have bi-directional integration. This situation is very normal in an evolutionary architecture because we may have multiple versions of functionally running concurrently from time to time as we upgrade a capability. We first covered this concept in Chapter 5, *Turning the Cloud into the Database*, where we discussed building it into our systems as a first-class capability so that we are ready to evolve.

When we are migrating from a legacy system, we will certainly have bi-directional integration. However, it is unlikely that the legacy system will have a latching feature, so we will need to add it non-invasively. We can accomplish this by storing *cross-reference* and *latching* information in the `entities` table that we introduced in the *Enriching data* section.

discriminator	pk	sk	latched
xref	subsys1|product|1	extsys1|prod|a	closed
xref	subsys1|product|2	extsys1|prod|b	open
xref	subsys1|product|3	extsys1|prod|c	open

Table 7.1: Cross-reference and latching

In this table, we store a cross-reference between the internal and external identifiers of the entity. For example, the **partition key** (**pk**) of a DynamoDB table would contain the internal `namespace|type|id`

combination, and the **sort key** (**sk**) would contain the external `namespace|type|id` combination. The row will also contain a **latched** field. When the ESGs in *Figures 7.12* and *7.13* synchronize data between the systems, the following additional logic will occur:

1. The *egress* Transform ESG in the Direct SQL flow will set **latched** to **closed** when it sends an update to the legacy system.
2. The update to the legacy data will trigger the CDC flow, but the *ingress* Transform ESG will see that **latched** is **closed**, so it will short-circuit and just set **latched** back to **open**.
3. Later on, when a user performs an update in the legacy system, the *ingress* Transform ESG in the CDC flow will not short-circuit and will publish an external domain event.

Slow data resync

We touched on the concept of fast versus slow data before, in *Chapter 5, Turning the Cloud into the Database*. Fast data is transactional data that produces events continuously, whereas slow data is reference data that is essentially static and produces events infrequently. This means that the events for slow data are spread throughout the event lake, which makes it more tedious to replay those events.

We can solve this problem by periodically publishing resynchronization events about the current state of the slow entities. This allows downstream services to ensure they are up to date and also records these events in the event lake, so they are easily included during a replay. We can also think of this as similar to creating a *snapshot* in *event sourcing*.

During a migration, the slow data will typically live in the legacy systems. After a migration, much of this data will typically live in an upstream

subsystem, following the data life cycle approach for dividing up subsystems, as we discussed in *Chapter 2*, *Defining Boundaries and Letting Go*. In either of these cases, it may be beneficial to build a resynchronization capability into an *ingress* ESG so that your team does not have to rely on coordinating with an upstream team to initiate a resynchronization in an emergency situation. *Figure 7.24* depicts the resources involved:

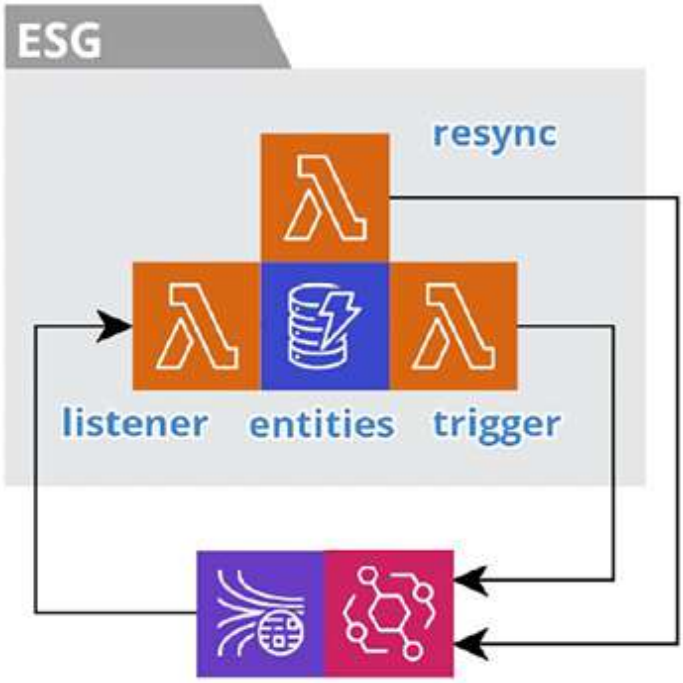


Figure 7.24: Ingress slow data resync

The `listener` function of the *ingress* ESG consumes external domain events, transforms them into the internal format, and stores them in the entities table following the *single table pattern* that we covered in *Chapter 5*, *Turning the Cloud into the Database*. The `trigger` function produces internal domain events when an entity changes. Note that this provides a layer of idempotence, in the case that an upstream system publishes the same event multiple times or if it performs a resynchronization itself.

The `resync` function scans the `entities` tables and produces internal domain events, such as `thing-resync`, for all the entities. This includes deleted entities as well, with a `deleted` flag, following the *soft delete* approach covered in *Chapter 5*, *Turning the Cloud into the Database*. This allows downstream services to bring themselves up to date if they have become out of sync.

Summary

In this chapter, you learned why it is important to create an anti-corruption layer between your internal system and external systems so that changes in external systems have minimal impact on the internals of your system. You learned about the ESG pattern and how to use it to integrate with other systems.

We dug into the many variations on the ESG pattern, and you learned how to handle ingress and egress flows with third-party and legacy systems, how to create a system topology of autonomous subsystems, and how to integrate across cloud providers. You also learned how to tackle common data challenges.

In the next chapter, we will cover the Control service pattern and see how it facilitates collaboration between BFF and ESG services. We will look at variations for orchestration, complex event processing, event sourcing, compensation, and more.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow

the QR code below:

https://discord.gg/B8TubSxc35

8

Reacting to Events with More Events

We covered the boundary service patterns in *Chapter 6*, *A Best Friend for the Frontend*, and *Chapter 7*, *Bridging Intersystem Gaps*. These services work at the boundary of the system to support end users and external systems, and they are responsible for catering to the kinds of changes driven by these actors.

Now, we turn our attention to the inter-service collaborations that represent the business policies and control logic of a system. We will learn how to create *control services* that are responsible for catering to the kinds of changes driven by the actors who own these business policies.

In this chapter, we're going to cover the following main topics:

- Promoting inter-service collaboration
- Dissecting the Control Service pattern
- Orchestrating business processes
- Employing the Saga pattern
- Calculating event-sourcing snapshots
- Implementing **complex event processing** (**CEP**) logic
- Leveraging **machine learning** (**ML**) for control flow

Promoting inter-service collaboration

In *Chapter 1, Architecting for Innovation,* we saw that the role of architecture is to enable change so that teams can continuously experiment and uncover the best solutions for their users. We enable continuous change by defining fortified boundaries around things that change together so that we can control the scope and impact of any given change. The key to defining these boundaries is to understand the driving force behind change.

In *Chapter 2, Defining Boundaries and Letting Go,* we found that people (that is, actors) are the driving force behind change. We identified a set of *autonomous service patterns* that support the different kinds of actors so that each service is responsible to a single actor. In this chapter, we will dig into the details of the **Control Service** pattern.

Control services work between the boundary (that is, **Backend for Frontend** (**BFF**) and **External Service Gateway** (**ESG**)) services as *mediators*, to promote collaboration between the different actors. They embody the rules and business logic that constitute the policies of the *organization*. These policies are apt to change frequently as the business adapts to its dynamic environment. Thus, control services are responsible to the stakeholders who own these policies.

Choreography’s pros and cons

One way to understand the value of control services is to look at the pros and cons of the alternative. In an event-driven system, services collaborate by exchanging events. Choreography is by and large the most common form of inter-service collaboration. It is characterized by the lack of a central coordinator. Upstream services emit events, and downstream services react and produce more events. We design and assemble a sequence of these interactions, and the individual services control the flow.

The following diagram depicts a choreography that promotes the flow of data through a system so that we can materialize data in downstream services. Following our food delivery example, when a restaurant updates its menu, other services react to keep their cache synchronized:

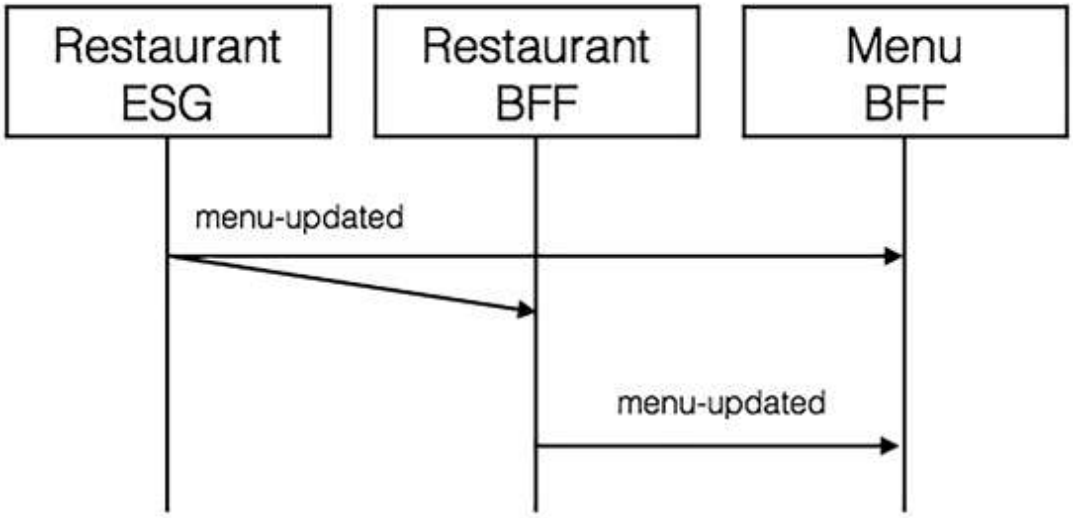


Figure 8.1: Data flow choreography

The restaurant's system invokes the external interface provided by **Restaurant ESG** to produce a **menu-updated** event. The **ESG** does not explicitly know what downstream services will consume the event. We designed the **Menu BFF** to react to this event so that customers can order from the most up-to-date menu. The **Restaurant BFF** enables system administrators to make menu corrections. As such, this BFF will both consume and produce the **menu-updated** event.

This data flow example highlights some of the benefits of the choreography approach. Upstream and downstream services are not explicitly coupled to each other. Upstream services can publish an event and do not expect a response. We can add or remove downstream services over time, and multiple services can play similar roles. Any number of services can

participate in the collaboration, so long as they uphold the event contract. **Choreography is the best approach for implementing these data-synchronization flows.**

The following diagram depicts a choreography that promotes the **flow of control** through a system so that multiple services can collaborate to perform the steps of a business process. Following our food delivery example, when a customer submits an order, the restaurant needs to receive the order and communicate that the meal is being prepared:

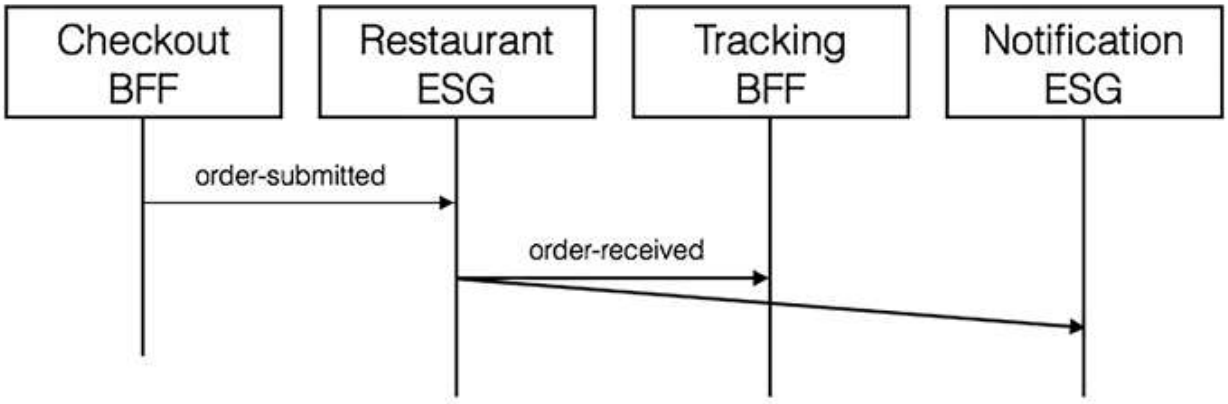


Figure 8.2: Control flow choreography

Checkout BFF produces an **order-submitted** event. The **Restaurant ESG** *egress* flow reacts by invoking the endpoint provided by the specific restaurant. The restaurant's order system receives the order, and then it invokes the endpoint for the **Restaurant ESG** *ingress* flow to send confirmation and produce an **order-received** event. **Tracking BFF** and **Notification ESG** each consume the **order-received** event and make the information available to the customer.

In this control flow example, the choreography approach works perfectly well up to this point. But the business process is far from over, and things get much more complicated from here. For example, the **order-submitted** event

also needs to trigger the selection of a driver, and we need to track the driver's status. It might be straightforward to use choreography for this happy path as well, but the driver functionality becomes much more complicated as we include all the alternate paths and error paths.

It wouldn't be reasonable to put all this logic in **Driver BFF**. **Driver BFF** will change with the needs of the drivers, and the functionality for selecting drivers will change with the needs of the business. Furthermore, **Tracking BFF** and **Notification ESG** should just be *observers* of the driver functionality and consume a generic status event that shields them from the details of how the status is determined. Thus, a service that sits in the middle and acts as a mediator between the boundary services may be a better approach when the business logic is more intricate.

Choreography has the advantage of being straightforward to implement. It is perfect for *data synchronization and observation flows*. It works well for basic control flow scenarios but may not scale well for more complex scenarios. As the highly interrelated control flow logic spreads across the system, choreography becomes harder to understand, trace, debug, test, and maintain.

Instead, we can move control flow logic upstream and consolidate it in dedicated control services. These services consume lower-order events, evaluate rules, and produce higher-order events. They come in many variations, such as business process *orchestration*, *sagas*, *CEP*, *ACID 2.0 calculations*, and more. It is reasonable to start with choreography and evolve to control services as the optimal solution becomes clear. Ultimately, a solution may warrant a mixture of approaches.

Before we dive into some applications of this pattern, let's look at the anatomy of a control service.

Dissecting the Control Service pattern

Up to this point, we have learned how to identify the actors of a system and create boundary services for the end users and external system actors. BFF services produce events as the users perform actions. ESG services bridge events between systems and invoke external actions. We use choreography to synchronize data across the services to create the inbound bulkheads that protect the services from upstream failures.

In the early days of a system, it is typical to use choreography to implement the control flows of the system's business processes, but as the system evolves and matures, it becomes beneficial to *refactor these control flows into control services*. The following diagram depicts the resources that make up a typical control service:

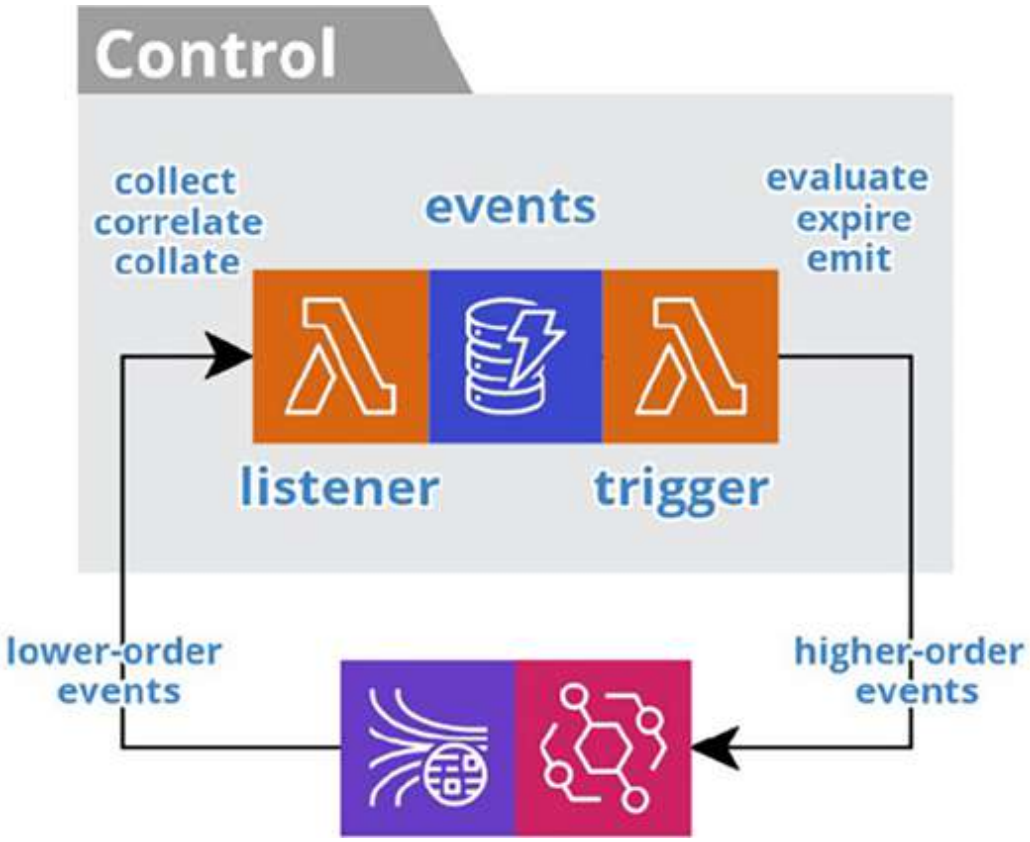


Figure 8.3: The Control Service pattern

Control services act as *mediators* between collaborating boundary services. They only consume and produce events; they do not expose a synchronous interface. The job of a **listener** function is to **collect**, **correlate**, and **collate** lower-order events in a micro **events** store. The job of a **trigger** function is to **evaluate** rules against the collected and correlated events and emit higher-order events that constitute the appropriate reaction to the identified conditions. They also implement time-based logic and produce expired events.

In essence, *control services react to events by producing more events*. We can think of these services as implementing a continuous **action-event-reaction loop**. The flow of arrows in *Figure 8.3* implies this loop. Boundary services produce events and control services consume them and produce

more events, then boundary services react to these events, and the cycle continues. Control services may even consume and react to their own events. The events that control services emit represent **higher-order** facts **that they** infer **from lower-order facts**. They record these facts with the *stream-first atomic action* of publishing an event so that downstream services can take responsibility for reacting.

It's a recursive implementation of the **inversion of responsibility principle** (**IRP**), with boundary services delegating *decision logic* to control services, and control services delegating *action logic* to boundary services. Each control service embodies a cohesive set of rules.

You can find a control service template project here: https://github.com/jgilbert01/templates/tree/master/template-control-service. Let's dig into the different kinds of rules.

collect

The job of a control service's `listener` function is to collect events of interest and store them in a micro events store. The purpose of this is twofold: *idempotence* and *correlation*.

Storing events in a serverless database, such as Amazon DynamoDB, helps to protect the control logic when upstream services produce duplicate events. As we covered in *Chapter 5, Turning the Cloud into the Database*, these databases implicitly provide idempotence because they do not send duplicate records to their **change data capture** (**CDC**) stream.

Following the single table design, the **primary key** (**pk**) field is set to `event.id`, the `sort key` (`sk`) field is set to the `EVENT` constant, and the

contents of the event are stored in the body of the record. The logic to collect events is reusable and a great candidate for declarative rules.

The following example builds on the concept of *pipeline patterns*, which we covered in *Chapter 4, Trusting Facts and Eventual Consistency*:

```
import { collect } from 'aws-lambda-stream'; const RULES = [{ id:
'r1', pattern: collect, eventType: [ 'order-submitted', 'order-
received', 'driver-assigned', . . . ], }];
```

Collection rules are very straightforward, as only a list of event types is required. Events of the specified types will be inserted into the micro events store, and all other events will be ignored. From here, things start to get more interesting as we address *correlation*.

correlate

We consider some rules as *simple* because they only need to evaluate the state of a single event. We consider other rules as *complex* because they need to evaluate the state of multiple related events. Storing events in a micro events store gives us the opportunity to correlate related events so that we can easily retrieve them when we need to evaluate a complex rule.

By default, the `collect` pipeline uses the `event.partitionKey` value as the correlation key. This is based on the good assumption that we are using the partition key to route related events to the same stream shard. Following the single table design, covered in *Chapter 5, Turning the Cloud into the Database*, we set the `data` field with the correlation key. Later, when an event triggers a complex rule, we use the correlation key stored in the `data` field to retrieve all the related events with a single query. This query leverages a global secondary index that we configure with the `data` field as the `primary key`.

Sometimes, there is more than one way to correlate events. In this case, we can declare additional `correlate` rules in the `trigger` function. The `collect` rule still handles idempotence, and once an event is inserted, the `trigger` function will create additional correlations.

In the following example, a set of `order` events are correlated by `customerId` instead of `order.id`. This will allow the evaluation of conditions across multiple orders instead of a single order:

```
import { correlate } from 'aws-lambda-stream'; const RULES = [{
id: 'r2', pattern: correlate, eventType: ['order-submitted', . .
.], correlationKey: 'order.customer.id', }];
```

To support multiple correlations, we use the main index of the table with the `primary key` (`pk`) field set to `correlationKey`. When these records trigger an evaluation rule, we use the correlation key stored in the `pk` field to retrieve all the related events with a single query.

collate

Order tolerance is another important benefit of *collecting* and *correlating* events in a micro events store. As we covered in *Chapter 4*, *Trusting Facts and Eventual Consistency*, when everything is running smoothly, the throughput of the system benefits from events arriving in the correct order; however, our services must have the resilience to accommodate situations where events arrive in the wrong order.

To support order tolerance, a service needs to take into consideration all the work that has already been performed and recognize when it has not performed certain work. For example, if a **cancellation** event arrives before an **initialization** event, then the logic will need to decide whether to defer reacting to the **cancellation** event or proceed and ignore the **initialization**

event when it arrives later. Collecting events in a micro events store allows a service to remember which events it has already seen. A service can even consume its own events to record which actions it has already taken.

Collating the events in the micro events store allows us to retrieve the correlated events in chronological order. This increases the efficiency of the rules as it minimizes the programmatic sorting that each must perform. To accomplish this, a service needs to choose the best field to use as the *sort key* for any *indexes*. The typical choices include **event.timestamp**, stream **sequenceNumber**, and **event.id**.

The `event.timestamp` is frequently the best option, so long as it does not result in duplicates, because it represents the time that a domain event occurred. The `sequenceNumber` is certain to avoid duplicates, but it represents the order in which events were added to the stream and not necessarily the order in which they happened. The `event.id` will also avoid duplicates and may be a good candidate if it contains **Version 1** (**V1**) **universally unique identifiers** (**UUIDs**), as they are timestamp-based.

To fully support order tolerance, it is necessary to enumerate all the potential scenarios and declare rules that account for all the combinations. We will see how *decision tables* can help in the *Implementing complex event processing (CEP) logic* section. For now, let's move on and look at *evaluation* rules.

evaluate

Every time a new event is inserted into the micro events store, the `trigger` function has another opportunity to assess the state of the system based on the current state in its micro events store and determine how to react. This is accomplished by consuming from the CDC stream of the database, such as

an Amazon DynamoDB stream, and *evaluating* rules against the incoming and correlated domain events.

The overall flow of evaluating events and reacting is a great candidate for declarative rules.

The following example builds on the concept of *pipeline patterns*, which we covered in *Chapter 4, Trusting Facts and Eventual Consistency*:

```
import { evaluate } from 'aws-lambda-stream'; const RULES = [{ id:
'r3', pattern: evaluate, eventType: ['order-submitted'], filters:
[(uow) => uow.event.order.category === 'xyz'], emit: . . . }];
```

The first step in the `evaluate` flow is to match rules to the incoming domain events based on their `eventType` value. In many cases, this may be all that is necessary to proceed to the outcome of a rule. It may also be necessary to filter based on the content of an event. In this first example, the rule tests the content of the `order.category` field. We consider this a *simple* rule because it only evaluates conditions against the incoming domain event.

We consider the next example as a *complex* rule because it needs to evaluate conditions against multiple correlated domain events. This rule applies to two event types, `order-received` and `driver-assigned`, and the condition is that both events must be present before the rule reacts. This condition is declared as an expression that contains a block of code. The presence of the expression directs the rule to retrieve the `correlated` events before invoking the `expression` function:

```
import { evaluate } from 'aws-lambda-stream'; const RULES = [{ id:
'r4', pattern: evaluate, eventType: ['order-received', 'driver-
assigned'], expression: (uow) => { const found =
uow.correlated.filter((e) => ['order-received', 'driver-assigned']
.contains(e.type)); if (found.length === 2) { return found; } else
{ return false; } emit: . . . }];
```

In this particular example, the order of the events does not matter. If the `driver-assigned` domain event arrives first, then the condition will not be met. When the `order-received` event does arrive, then the condition is met and the rule reacts. The same will hold true if the domain events arrive in the opposite order. In many cases, such as this one, it is straightforward to account for *order tolerance* within a single rule. In other cases, it is better to define multiple rules with different guard conditions to account for the various scenarios.

Note that in the two preceding examples, the conditional logic is implemented inline for brevity. In practice, it is better to implement and test the conditions as separately declared functions. As a control service grows, these functions will begin to form a **domain-specific language** (**DSL**) with reusable code fragments that we can combine in different ways to account for the many scenarios.

These two examples represent just one possible way to implement the *evaluate* phase of the Control Service pattern. We will look at some other variations later in the chapter, but before we move on, we need to look at the intended outcome of a control service, which is to produce more events.

emit

The purpose of a control service is to react to the *lower-order* events produced by upstream services and produce *higher-order* events to control the flow of downstream services. These higher-order events provide a layer of indirection that helps to decouple downstream services from upstream services. The logic that controls when we emit these higher-order events also tends to change independently of the upstream and downstream services, which reinforces the value and purpose of the Control Service pattern.

A *higher-order* event represents the *fact* that a specific condition was met, and it is the job of a control service to record these facts. In *Chapter 4*, *Trusting Facts and Eventual Consistency*, we covered the importance of *atomic actions* in eventually consistent systems. In control services, we leverage **stream-first** atomic actions to persist the facts. Once a fact has been persisted to the stream, the control service is free to continue its *action-event-reaction loop*, and it defers to downstream services to take responsibility and react accordingly.

The following code fragment completes the preceding example by adding the `emit` logic. Again, the code is included inline for brevity:

```
const RULES = [{ id: 'r4', . . . emit: (uow, rule, template) => ({
...template, type: 'delivery-initiated', order: uow.event.order,
}), }];
```

The `template` argument contains default values, including a unique and deterministic event ID, plus a timestamp, partition key, tags, and references to the events that triggered this new event. We can adorn this template with additional data, such as the domain object and the event type to emit. Now, let's look at expiration rules.

expire

Following on from the data life cycle architecture philosophy covered in *Chapter 5*, *Turning the Cloud into the Database*, the control service's micro events store should remain lean and retain events for only as long as necessary. We leverage the **time to live** (**TTL**) feature of the database, such as Amazon DynamoDB, to remove events after they are no longer needed. For example, a default TTL of 33 days is likely sufficient for most business scenarios, while also ensuring idempotence in the case of event replays.

It is important to note here that control services are very time-sensitive, which has implications for replaying events to them. We should take caution when replaying events that are older than the TTL. In this case, it is often more appropriate to replay the events of the control service to downstream boundary services.

We can also use the TTL feature to implement *dynamic expiry* logic. The following example contains rules that will trigger an escalation when a step in a business process is taking too long:

```
const RULES = [{ id: 'r5', pattern: timer, eventType: ['order-
submitted'], ttl: (uow) => uow.event.timestamp + 600, // 10
minutes },{ id: 'r6', pattern: expired, eventType: ['order-
submitted'], expression: (uow) => uow.correlated.filter((e) =>
['order-received'].contains(e.type)).length < 1), emit: 'order-
submission-expired', }];
```

Following our food delivery example, once the `order-submitted` event is produced, we expect the restaurant to respond with an `order-received` event within 10 minutes. The `timer` rule creates a record with the calculated `ttl`. After the period expires, the `expired` rule produces an `order-submission-expired` event to trigger a downstream escalation process. The escalation logic will likely be part of the same control service, but this rule is focused on recording the fact that the time has expired.

Note that the **service-level agreement** (**SLA**) for a database's TTL feature may be low, which means that this approach is most applicable for *event-time*-based logic instead of *processing-time*-based logic. For example, a slight delay in emitting an `order-submission-expired` event is probably tolerable, or using expiration to calculate end-of-month totals could tolerate a much longer delay, so long as it eventually happens.

We must also pay attention to order tolerance when implementing time-sensitive logic—for example, an `order-received` event could be delayed by an error condition, but the restaurant is working on the order. So, when the `order-received` event is ultimately received, there may be a need to compensate for an escalation that was already triggered by an `order-submission-expired` event. We will be covering compensation transactions in the *Employing the Saga pattern* section.

This kind of order tolerance logic is tedious but it is where many systems fall short and get into trouble. It is also where the Control Service pattern and the rule-driven approach come in very handy.

As I mentioned previously, declarative rules are just one way to apply the Control Service pattern. In some cases, it may be best to have a completely bespoke implementation but the collect, correlate, collate, evaluate, emit, and expire concepts will still be applicable.

Now that we are familiar with these concepts, let’s look at some different ways to apply the pattern.

Orchestrating business processes

A *business process* is a long-lived flow of activities executed in a specific sequence to achieve the desired outcome. These activities consist of human tasks and atomic actions. The duration of a business process can range from minutes to days, or longer, depending on the nature of the human tasks that must be performed. For example, the food delivery process, which involves preparing and delivering food, should be measured in terms of minutes,

whereas a typical business process that requires a management approval step may involve waiting hours or days for a manager to approve a task.

There are two approaches to implementing business processes: *choreography* and *orchestration*. These terms are borrowed from the arts and are used as metaphors for software techniques—for example, a team of dancers works together to perform a choreographed set of movements but the choreographer is not in control of the actual performance, whereas an orchestra is led by a conductor as it performs a musical composition.

As we have seen, there are limits to the *choreography* approach and it is only recommended for the most basic of business processes. *Orchestration* is the preferred approach. Orchestration is distinct from choreography as the flow of control is managed centrally by a conductor or mediator. This is the role that a *control service* plays. We can model the logic of a control service (that is, a business process) as a state machine using an activity diagram, such as the one shown in the following figure:

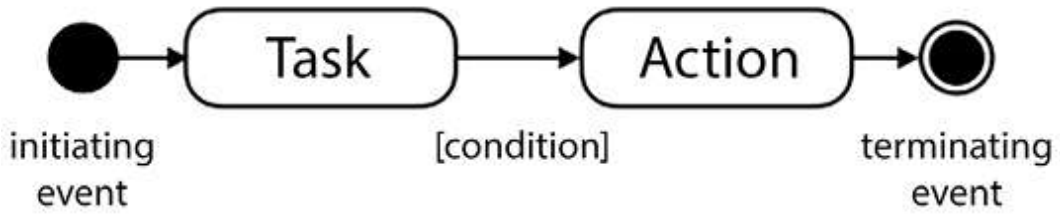


Figure 8.4: Activity diagram

A process is initiated by a specific event, and the end of the process is signaled by another event. Each step in the process represents a different state, indicating when a specific activity—such as a human task or an atomic action—should be performed. The directed lines between the states represent the state transitions and their guard conditions.

A *control service* is responsible for orchestrating (that is, mediating) state transitions, but not for implementing the individual activities, which are instead implemented by collaborating services—for example, a BFF service will implement a human task, and an ESG service might invoke an atomic action on an external system. Let's take a deeper look at how we implement state transitions with entry and exit events.

Entry and exit events

If you are familiar with traditional **business process management** (**BPM**) tools, such as AWS Step Functions, then you will notice a difference in how control services orchestrate business processes. For example, to perform an atomic action step, a traditional BPM tool will directly invoke other services and wait for a response. It may also support publishing an *entry event* when the step starts and an *exit event* when a response is received so that others can observe the flow of control, but these events are completely optional.

Of course, having one service directly invoke another service violates our goal of creating autonomous services with fortified boundaries.

Control services, on the other hand, implement orchestration using just *entry and exit events*. This approach eliminates synchronous invocation dependency and builds on the **dependency inversion principle** (**DIP**), the **inversion of responsibility principle** (**IRP**), and the **substitution** principle to create a completely decoupled solution. It is also implicitly supporting observers, such as an event lake, without additional effort.

Following this approach, the control service acts as a policy-setting module that establishes the contracts that other services must implement to participate in the business process. These contracts are the pairs of entry and exit events. The following sequence diagram provides an example of these

events for a process that is similar to the one depicted in *Figure 8.4*. In this example, a user wants to perform an action, but approval is required first:

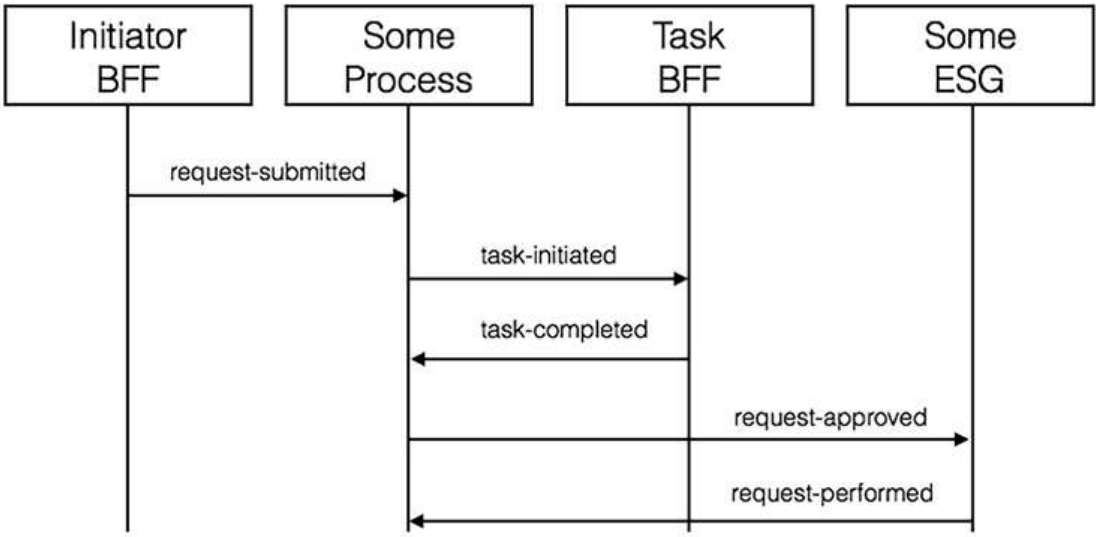


*Figure 8.5: Orchestration sequence diagram The **Initiator BFF** service signals its intention to start a business process by publishing a **request-submitted** event. The **Some Process** control service transitions to the **Task** state when it receives the **request-submitted** event and emits a **task-initiated** event to indicate that it has entered that state. To exit this state, the process expects to receive a **task-completed** event.*

The **Task BFF** service implements the human activity. It consumes **task-initiated** events and presents tasks to users in a work list. When a user completes a task, it produces a **task-completed** event with the user's disposition.

The **Some Process** control service receives the **task-completed** event, evaluates the conditions, and determines whether to transition to the next state. If the user completed the task with an approval, then the business process transitions to the **Action** state and emits a **request-approved** event to indicate that it has *entered* that state.

To *exit* this state, the control service expects to receive a **request-performed** event.

The **Some ESG** service consumes the **request-approved** event, performs the atomic action, and produces a **request-performed** event when it has confirmation that the action is complete. At this point, the process is complete. The **Some Process** control service collects the **request-performed** event but has no further reaction.

The following code block shows how a control service can implement the business process using a set of rules. The first rule emits the `task-initiated` event in reaction to the `request-submitted` event. The second rule emits the `request-approved` event in reaction to the `task-completed` event. These rules are where we wire the outputs of one activity to the inputs of another activity as well—for example, in the first rule, `uow.event.request` is mapped to `task.artifact`, and in the second rule, `uow.event.artifact` is mapped to `request`:

```
import { evaluate } from 'aws-lambda-stream'; const RULES = [{ id:
'o1', pattern: evaluate, eventType: ['request-submitted'], emit:
(uow, rule, template) => ({ ...template, type: 'task-initiated',
task: { subject: `Review request: ${uow.event.request.id}`, role:
'Reviewer', artifact: uow.event.request, }}), }, { id: 'o2',
pattern: evaluate, eventType: ['task-completed'], filter: (uow) =>
uow.event.task.outcome === 'approved', emit: (uow, rule, template)
=> ({ ...template, type: 'request-approved', request:
uow.event.artifact, }), }];
```

Leveraging entry and exit events allows us to completely decouple the various collaborators—for example, the **Initiator BFF** service does not know or care which business process handles the request, and the **Some Process** control service can be initiated by any service. The control service, in turn, does not know or care which service reacts to an entry event, so long as it emits the expected exit event. These are examples of the flexibility

provided by the IRP, as upstream services defer to downstream services that take responsibility for reacting to the upstream events.

Following the DIP, the high-level control service is only concerned with when to emit an entry event and not with how to react. It establishes the contract of each *entry/exit* event pair and delegates the details to the low-level boundary services, and—as just mentioned—the control service has no dependency on which services implement the activities.

Following the **Liskov substitution principle** (**LSP**), we can substitute different collaborators—for example, a legacy system may implement an activity until the capability has been re-architected. Multiple collaborators can also participate simultaneously, with each filtering for and reacting to a subset of the events. This can be useful for beta-testing different potential implementations or for supporting different needs under different scenarios. In any case, the important aspect is that these details can change independently of the control service that defines the business process.

The low-level boundary services are also not coupled to the actual business process and can potentially participate in multiple business processes. For example, a boundary service such as the **Task BFF** service can define a generic pair of *entry/exit* events that it supports, such as **task-initiated** and **task-completed**, and thus can participate in any business process that uses those events.

An activity in a business process can also be implemented by another control service that is essentially implementing a sub-process. In this case, the *entry* event would *initiate* the sub-process, and the *terminating* event of the sub-process would be the *exit* event expected by the parent process.

Ultimately, the most important characteristic of orchestrating business processes with control services is that we are implementing all the control-

flow logic for the business process in one place. This has three key benefits, as follows:

- It makes learning and understanding the policies easier because they are not spread across the system.
- When the policies change, the impact has a higher likelihood of being limited to a single control service.
- The policies can be tested in isolation from the details of the boundary services.

Now, let's look at how to implement more complicated business processes with parallel execution paths.

Parallel execution

The previous business process example is considered to be a simple process because the steps in the process are executed sequentially. More complex business processes include parallel execution. Parallel execution can take the form of *fan-outs* or *forks and joins*.

A **fan-out** is applicable when multiple activities can be executed in parallel but there is no need for all the activities to be completed before continuing on to the next activity. For example, at a certain point in the flow, it may be necessary to send the customer a status update, but there is no need to hold up the rest of the process while this happens.

The following diagram depicts how this might look, with **Step 1** fanning out to **Step 2** and **Step 3**. **Step 2** transitions to **Step 4** when it is complete, whereas **Step 3** has no follow-on steps. Implementing a fan-out is straightforward. We only need to emit an entry event, and there may not be a need to react to an exit event:

Figure 8.6: Fan-out

This is also an opportunity for multiple services to react in parallel to the same entry event, such as one ESG service sending a status update by email and another ESG service sending it by **Short Message Service** (**SMS**). Each will emit an exit event for auditing purposes, but the process does not need to react to these events.

Forks and joins are applicable when we need to execute multiple activities in parallel and they all must be completed before continuing to the next activity. For example, in our food delivery system, sending an order to the restaurant and selecting a driver can happen in parallel, but we may not want to dispatch the driver to the restaurant until we have confirmation that the order was received.

The following diagram depicts how this might look. The first black bar represents the *fork*, with **Step 2** and **Step 3** executing in parallel. The second black bar represents the *join*, which must occur before transitioning to **Step 4**. To implement a join, we leverage the correlated events that the control service keeps in its micro events store. The join rule asserts the presence of all required exit events before emitting the next entry event:

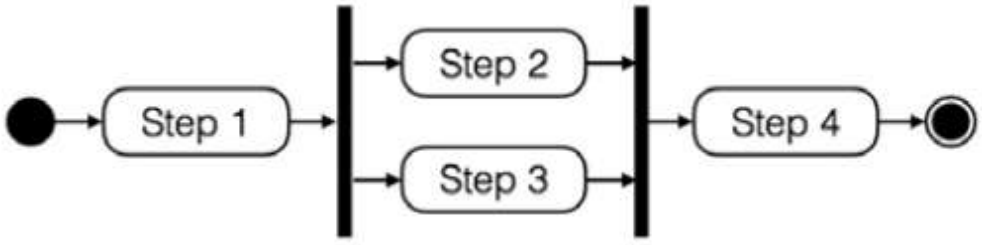


Figure 8.7: Fork and join

The following code block shows how a control service can implement a join using rules. We have already seen a similar example of CEP in the *Dissecting the Control Service pattern* section. The `join` utility function asserts whether all the listed events have been correlated in the micro events store. When each step is completed, the rule has an opportunity to evaluate whether the other step has been completed. The entry event for *Step 4* is emitted once the expression evaluates to `true`:

```
import { evaluate } from 'aws-lambda-stream'; import { join } from
'./utils'; const RULES = [{ id: 'o3', pattern: evaluate,
eventType: ['step2-completed', 'step3-completed'], expression:
join(['step2-completed', 'step3-completed']), emit: 'step4-
initiated', }];
```

Parallel execution is just one example of how a business process can become complex. First-order alternate paths, such as a rejection flow in an approval process, are straightforward to model with activity diagrams. However, second-, third-, and *Nth*-order alternate paths are much easier to reason about as straight rules, with help from tools such as *decision tables*. One pattern for handling alternate paths is called the Saga pattern. Let's look at Sagas next, and we will cover decision tables in the *Implementing complex event processing (CEP) logic* section.

Employing the Saga pattern

In *Chapter 4, Trusting Facts and Eventual Consistency*, we addressed the reality that modern distributed systems no longer rely on distributed transactions because they do not scale well and because of the heterogeneity of the many technologies involved. Instead, we create resilient, eventually consistent systems by breaking long-lived transactions down into a sequence

of atomic actions. The outcome of each action is emitted as an event that is recorded as a fact. Downstream services react and the cycle repeats until the system reaches a consistent state in near real time.

However, we do need to account for scenarios when something goes wrong downstream and the long-lived transaction is not able to move forward to a consistent state. In these cases, we need to undo the sequence of actions performed up to the point of failure so that the system is returned to a consistent state. This pattern is referred to as a **Saga** and was first discussed by Hector Garcia-Molina and Kenneth Salem in their 1987 Princeton University paper entitled *SAGAS* (http://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf).

Let's dig into how we implement Sagas using compensating transactions and abort events.

Compensating transactions

Boundary services that participate in long-lived transactions may need to implement compensating transactions. These are atomic actions that **undo** the effects of a previous action. They may be as simple as flipping a status field or canceling an outstanding task for a human activity. Alternatively, they might need to redo a complicated calculation or create a new task for a human to perform a manual activity. The scope of a compensating transaction is only limited by the scope of the transaction it intends to reverse.

It is important to note that a compensating transaction is not akin to a rollback. **The objective is not to erase the evidence of the previous transaction.** The previous transaction is a historical fact, and the

compensating transaction will become a fact as well, with a clear audit trail in the event lake. An analogy is available in the field of public accounting, where nothing is ever erased and mistakes are corrected by adding adjusting journal entries to the standard general ledger.

Compensating transactions may be further complicated by the fact that multiple transactions may have interlaced and updated the same data since the previous transaction was executed. In this case, the effects of the subsequent transactions should not be undone. This may be as simple as subtracting the *incremental* value instead of setting a discrete value, or it may require keeping a *micro events store* so that the state can be recalculated.

It will also be necessary to account for the **cascading effect** of the events that will be produced by the compensating transaction. These events will provide the audit trail, but they may also cause downstream services to react. This will usually be the intended behavior but it is still necessary to reason about these impacts when designing the compensation algorithm.

Now let’s see how we can use abort events to signal the need for compensating transactions.

Abort events

A boundary service emits an *abort event* when it is unable to complete its work. Abort events are distinct from the concept of *fault events*, which we covered in *Chapter 4, Trusting Facts and Eventual Consistency*. Fault events signal that an unexpected error condition occurred, whereas an abort event signals that an expected condition occurred. A fault event can be resubmitted once the condition is corrected so that the system can move forward to reach a consistent state. Abort events, on the other hand, indicate that there is **no**

way forward and signal *upstream services* to perform their compensating transactions so that the system can be returned to a consistent state.

As with any other collaboration, we can implement a Saga using choreography or orchestration. The following sequence diagram provides an example of implementing the Saga pattern with *choreography*:

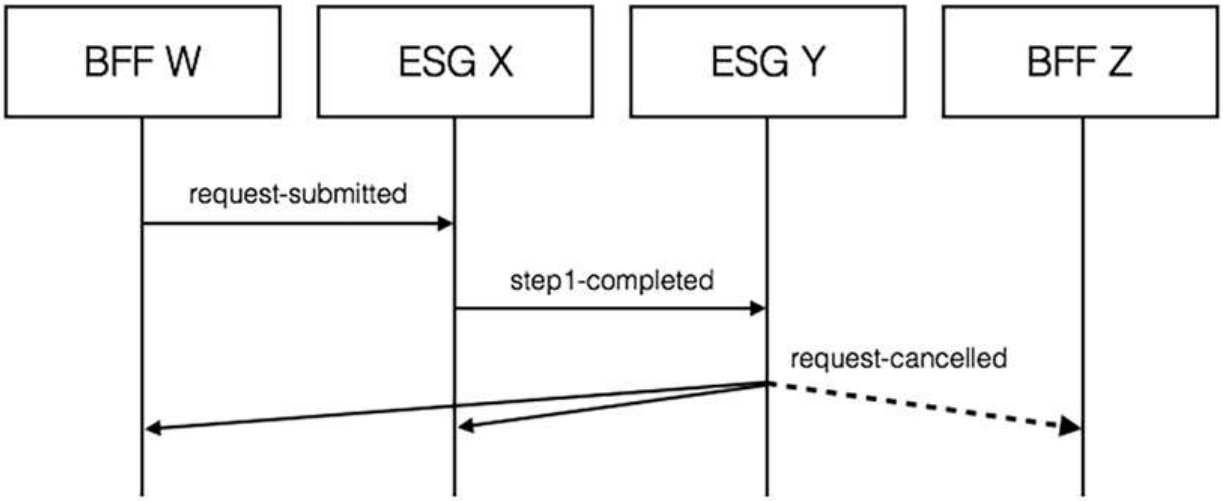


Figure 8.8: Choreography Saga

The **BFF W** service initiates a long-lived transaction with the **request-submitted** event. The **ESG X** service reacts and emits a **step1-completed** event. The **ESG Y** service reacts but is unable to complete its step and *aborts* by broadcasting a **request-cancelled** event. The **BFF W** and **ESG X** services react by performing their compensation transactions.

Broadcasting the **request-cancelled** event makes the **ESG Y** service's job easier because it doesn't worry about which services will react. However, this means that services that would have performed subsequent steps, such as the **BFF Z** service, will need to determine if they can ignore the abort event or not.

As always, the choreography approach is reasonable for the happy path, but it begins to lose its appeal as the alternate paths begin to emerge. In this case, all the services are coupled to the **request-cancelled** event for signaling an abort scenario. To alleviate this coupling, we can use an orchestration approach. The following sequence diagram provides an example of implementing the Saga pattern with *orchestration*:

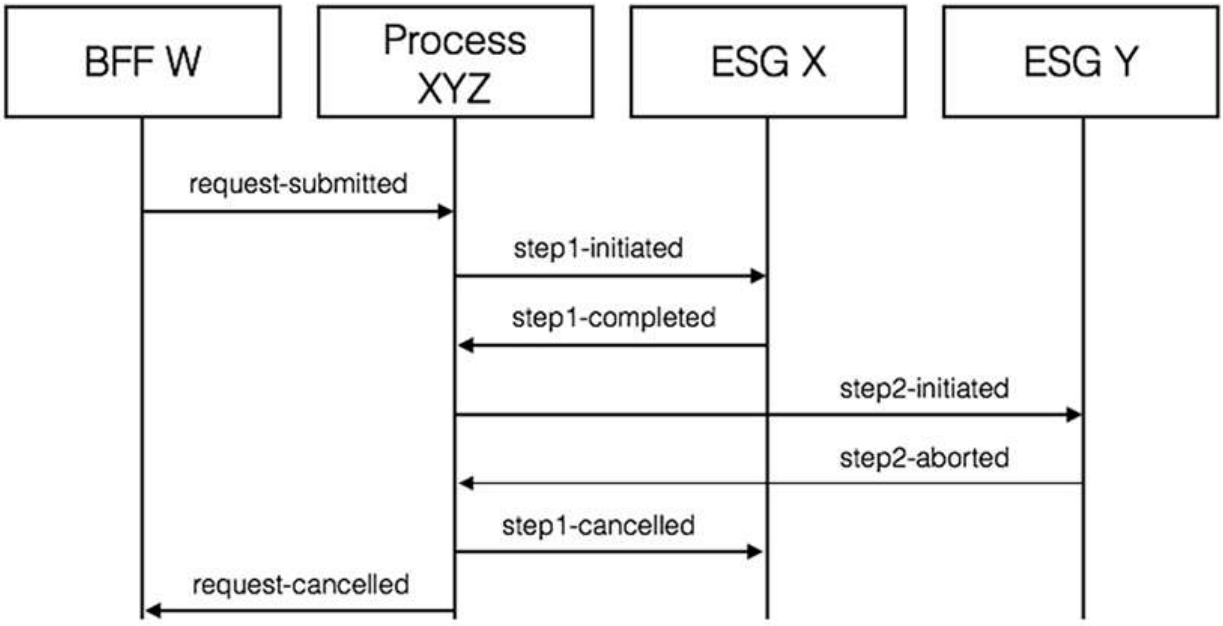


Figure 8.9: Orchestration Saga

Once again, the **BFF W** service initiates a long-lived transaction with the **request-submitted** event. The **Process XYZ** control service reacts by initiating *Step 1* and then *Step 2*. The **ESG Y** service is unable to complete its step and *aborts* by emitting a **step2-aborted** event. At this point, the **Process XYZ** control service knows exactly which compensating transaction needs to be performed and emits **step 1-cancelled** and **request-cancelled** events.

Here, we have expanded on the *entry and exit events* contract that we covered in the *Orchestrating business processes* section, by adding *abort*

and compensate events to the contract. Each collaborator has its events, which gives the control service precise control over which steps need to be compensated. This approach also makes it possible for boundary services to participate in multiple business processes.

The following code block shows how a control service can implement a Saga using rules. The first rule signals to the **BFF W** service to compensate when *Step 1* or *Step 2* aborts. The second rule signals to the **ESG X** service to compensate when *Step 2* aborts:

```
import { evaluate } from 'aws-lambda-stream'; const RULES = [{ id:
's1', pattern: evaluate, eventType: ['step1-aborted', 'step2-
aborted'], emit: 'request-cancelled', }, { id: 's2', pattern:
evaluate, eventType: 'step2-aborted', emit: 'step1-cancelled', }];
```

The complexity of the compensation transactions will vary by service, but these details are encapsulated by the services. This makes it easier to reason about the overall solution. Coordinating all the compensations that are necessary in a long-lived transaction is best managed with *orchestration* because these policies are managed in one place. All this improves our confidence in eventually consistent systems, and—more importantly—it improves our confidence about making changes to the system.

Now, let's turn our attention to another transactional approach known as ACID 2.0, and look at how we can use it to improve the event-sourcing pattern.

Calculating event-sourcing snapshots

We introduced **event sourcing** in *Chapter 4, Trusting Facts and Eventual Consistency*. Event sourcing is arguably the most important pattern in our

architecture because it turns events into facts. Instead of treating events as ephemeral messages, they become first-class citizens and live on in the event lake. They serve as an audit trail of activity within the system and enable the replaying of events to seed new services, repair ailing services, or help validate hypotheses for system improvements.

Unfortunately, event sourcing has a bad reputation for being overly complex for most use cases. This complexity revolves around the need to recalculate the current state from events over and over again to perform basic user scenarios.

But instead of abandoning event sourcing and all its benefits, it is better to employ approaches that eliminate the need to recalculate the current state.

Database-first event sourcing is a variation of the pattern we covered in *Chapter 5, Turning the Cloud into the Database*, and it gives us the best of both worlds. With this approach, we implement a BFF service in the traditional manner, whereby it only stores and interacts with the current state of domain entities in its database. On the other side of the database, the CDC stream emits change events, and a *trigger* function produces domain events to the event hub. This simplifies the development experience of the BFF service and improves performance for the user while maintaining the benefits of event sourcing for downstream services.

We can also improve the event-sourcing pattern by moving the calculation of the current state to *control services* in the middle of the system. As depicted in the following diagram, these control services collect detailed events, calculate the current state, and emit *higher-order snapshot* events. They also collect their own snapshot events and leverage them to optimize their calculations. Downstream BFF services use the snapshot events to easily

create materialized views for their users, without the complexity of calculating the current state themselves:

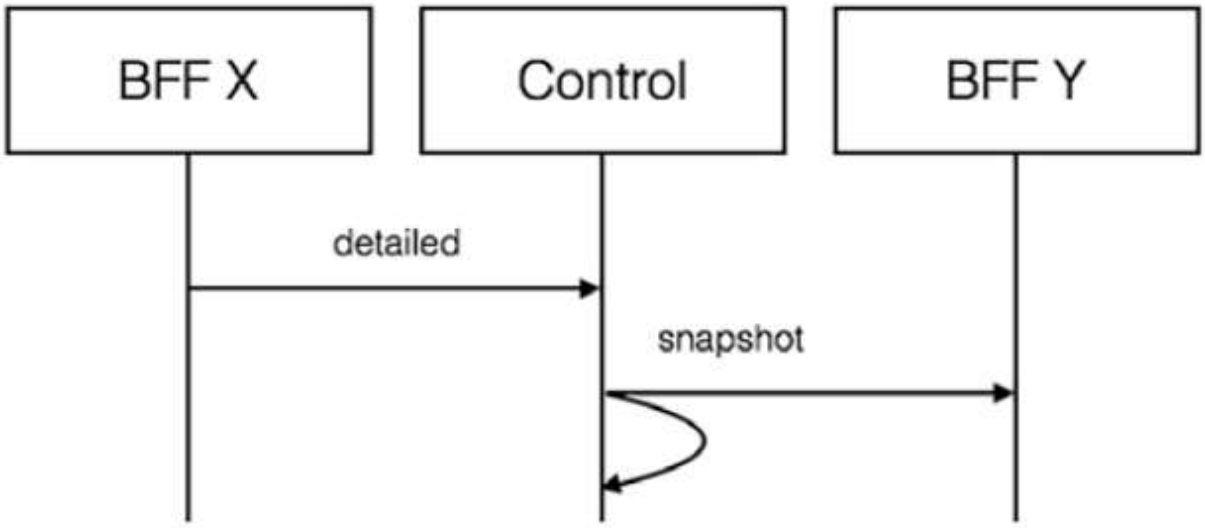


Figure 8.10: Snapshot sequence diagram

Let’s dig into snapshot events and look at how we leverage ACID 2.0 properties to calculate them.

What is ACID 2.0?

The realities of distributed systems and eventual consistency have inspired our industry to devise alternatives to distributed transactions. We have already covered the *Saga* pattern, which uses compensation transactions to return a system to a consistent state when there is a failure in a long-lived transaction. It is important to note that this is a guarantee provided by the application and not by the database itself. Instead, we, as architects, take responsibility for the eventual consistency of the system on the happy path and the failure paths.

Another transactional scenario that poses an interesting architectural challenge is when we have high contention for the same data. This includes scenarios that perform numerical calculations, such as maintaining account balances, tracking inventory, calculating bills, and accumulating metrics. The traditional approach, which locks these records and serializes the transactions, is not able to scale to high volumes or accommodate concurrent updates from distributed nodes.

To support these scenarios, we turn to an approach dubbed *ACID 2.0*. This new version of the classic acronym stands for *Associative, Commutative, Idempotent, and Distributed*. It was first documented by Pat Helland in his paper *Building on Quicksand* (https://arxiv.org/pdf/0909.1788.pdf), and he in turn credits Shel Finkelstein for the new definition of the acronym.

Many functional requirements exhibit these mathematical properties, so let's review them before leveraging them to improve the event-sourcing pattern, as follows:

- **Associative**—When performing a calculation, the result is independent of the grouping of elements.
- **Commutative**—When performing a calculation, the result is independent of the order of elements.
- **Idempotent**—When performing a calculation, the result does not change if we receive an element more than once.
- **Distributed**—Though not a mathematical property, the other properties must hold when we distribute the calculations across many nodes, such as across multiple regions.

These properties will be clearer in the context of an example. So, let's look at how these mathematical properties help us derive the current state of

domain entities from event sourcing.

Snapshot events

The performance of deriving the current state of a domain entity from event sourcing degrades as the number of events increases. To solve this problem, we record the results of a calculation as a snapshot so that we no longer need to recalculate the events that weighed into the snapshot. The next calculation starts with the latest snapshot and adds any additional events. This allows the performance of calculating the current state to remain reasonably consistent because we are limiting the number of events processed in the calculation.

A classic example of this process is tracking the balance of a bank account. We have **Deposit** events that increase the balance, **Withdraw** events that reduce the balance, and **Balance** events that provide a snapshot of the current balance. The following table demonstrates the ACID 2.0 properties. It includes two side-by-side examples of the same set of events:

Event Type'	Amount'	Event Type"	Amount"
Deposit	100	Deposit	100
Deposit	30	Withdraw	50
Balance	130	Balance	50
Withdraw	50	Deposit	30
Deposit	70	Deposit	20
Balance	150	Balance	100
Deposit	20	Deposit	70
Balance	170	Balance	170

Table 8.1: ACID 2.0 properties

The first thing to notice between the two examples is that the events arrived in different orders and the ending balance of both scenarios is **170**. This shows that the calculations are *commutative*. It is also the most important part because the system is eventually consistent, even if the intermediate results may vary. Next, notice that we calculated the **Balance** (that is, snapshot) events for different groupings of events, yet this did not impact the result. This is an example of the *associative* property.

We handle the *idempotent* property by the fact that a control service collects these events in a micro events store, and the trigger function that performs the calculations will not trigger multiple times if the same event is written more than once. For the *distributed* property, when we deploy a control service to multiple regions, an intermediate balance event in one region may not see an event in another region, but this is no different from an event being delayed and arriving out of order.

Now, let's look at what the code for calculating a snapshot event might look like in a control service. The service collects, correlates, and collates `deposit`, `withdraw`, and `balance` events in its micro events store and emits a new snapshot event when it receives a deposit or withdraw event. You can see the code in the following snippet:

```
const RULES = [{ id: 'c1', pattern: calculate, eventType:
['deposit', 'withdraw'], limit: 10, emit: (uow, rule, template) =>
{ const { balance, triggers } = calculateBalance(uow); return ({
...template, type: 'balance', account: { number:
uow.event.account.number, balance, }, triggers, }); }, }];
```

The `calculate` pipeline function retrieves the correlated events when a `deposit` or `withdraw` event arrives. It retrieves them in *reverse* order because we are only interested in the most recent events. We also limit the number of

events returned, since the older events are already counted in the latest snapshot. You can see the code in the following snippet:

```
const calculateBalance = (uow) => { const snapIdx =
uow.correlated.findIndex(e => e.type === 'balance'); const
snapshot = uow.correlated[snapIdx]; const latestEvents =
uow.correlated.slice(0, snapIdx); return { balance:
latestEvents.reduce((balance, e) => (e.type === 'deposit' ?
balance + e.deposit.amount : balance - e.withdraw.amount),
snapshot.account.balance ), triggers: [snapshot,
...latestEvents].map(({ id }) => ({ id })), }; };
```

The `calculateBalance` function finds the most recent snapshot and the latest events and then reduces them to the current balance. For brevity, this example focuses on the happy path and uses an overly simple slice that ignores edge cases such as when an older detailed event arrives after a snapshot. I'll leave these kinds of edge cases for you as a thought exercise, but note that the snapshot event includes the triggers so comparisons can be made. We could also have a pipeline that reacts to `balance` events, asserts the accuracy of the balance, and emits any adjustments. We can even do the same when the various events *expire* from the micro events store.

Next, we will look at using control services and CEP to initiate business processes as opposed to orchestrating them.

Implementing complex event processing (CEP) logic

Up to this point, we have mostly used control services to orchestrate business processes. In the previous examples, an end user initiated a business process via publishing an event from a BFF service. Humans are excellent complex-event processors. We naturally process all kinds of inputs

and quickly make decisions, but we aren’t always paying attention, or there may be too many inputs to reliably and consistently process them all.

This is where we can use control services to perform CEP and emit events to alert downstream services of their findings. The processing logic is considered complex because we are not just reacting to a single event. We collect and correlate **multiple events** and evaluate conditions across them for *actionable* insights. For example, we could audit a business process and assert that it is adhering to expectations. We have already seen a basic example of CEP when joining parallel paths in a business process.

Let’s look at how we can use decision tables to help reason about CEP rules, and then look at an important edge condition where we are missing events.

Decision tables

CEP is a powerful, effective, and clean approach for implementing business policies. However, reasoning about the correctness of these rules becomes more and more challenging as the number of events and conditions increases. This is because there are many more permutations to consider, especially when we account for compensation transactions and events arriving out of order.

Decision tables, such as the one shown here, are the recommended tools for modeling CEP rules. Each table presents a cohesive set of conditions:

Conditions	S1	S2	S3
Event A present	True	True	False
Event B present	True	True	False
Event C present	True	True	False
A.x <= B.y <= C.z	True	False	N/A
Actions			
Emit	audit-passed	audit-failed	N/A
Message		XYZ condition failed	
Notes			Nothing to process

Table 8.2: An example of a decision table The first column lists the conditions and the action to take when all the conditions are met. The remaining columns enumerate all the interesting permutations of the conditions. The cell values are True, False, and ***not applicable*** *(**N/A**). The bottom of the table declares the parameters to use when performing the actions.*

The major advantage of this approach is that it is easier to see when a condition or a permutation is unaccounted for. This can be particularly helpful when reasoning about events arriving out of order. Another advantage is that they are easily translated into the rules and corresponding test cases.

In this example, we are interested in three types of events. When all the events are present, then we can evaluate the given expression. If all the conditions are True, then we emit an `audit-passed` event; otherwise, we emit an `audit-failed` event with the given message. Some of the scenarios, such as `S3`, do not need to be translated into rules, but we should include them in the decision table for completeness. It is also helpful to include a Notes row in the decision table so we can provide supporting information.

Now, let's look at what the code for CEP might look like in a control service. The service *collects, correlates, and collates* `A`, `B`, and `C` events in its micro events store. As each of these event types arrives, the correlated events are

retrieved, and the conditions are *evaluated*. You can see the code in the following snippet:

```
const RULES = [{ id: 's1', pattern: evaluate, eventType: ['A',
'B', 'C'], expression: (uow) => { const { A, B, C } =
findAbcEvents(uow); return A & B & C & A.x <= B.y <= C.z; }, emit:
'audit-passed', }, { id: 's2', pattern: evaluate, eventType: ['A',
'B', 'C'], expression: (uow) => { const { A, B, C } =
findAbcEvents(uow); return A & B & C & !(A.x <= B.y <= C.z); },
message: 'XYZ condition failed', emit: auditFailed, }]; const
findAbcEvents = (uow) => ({ A: uow.correlated.find((e) => e.type
=== 'A'), B: uow.correlated.find((e) => e.type === 'B'), C:
uow.correlated.find((e) => e.type === 'C'), }); const auditFailed
= (uow, rule, template) => ({ ...template, type: 'audit-failed',
message: rule.message, })
```

The `findAbcEvents` helper function looks for all the event types in the `correlated` list. The first rule asserts that all three events are present and then checks that the additional condition evaluates to `True`. The second rule is similar but is asserting the inverse. We can think of it as the Else rule to rule s1. It also demonstrates that rules can include custom metadata, such as parameterizing the error message. The `auditFailed` function uses this metadata when creating the `audit-failed` event.

Now let’s see how we can handle scenario 3 if some events never arrive.

Missing events

An interesting nuance of CEP is knowing when you are done. For example, if you are auditing a business process and there are a dozen rules that all instances of the process must pass, then how do you know when an instance has passed all the rules? It is easy to know when a long-lived business process instance has failed because it only needs to fail one rule, but how do we know when all the rules have succeeded? We could wait until a process is

complete and then evaluate all the rules, but that may cause an unwanted latency in finding issues.

A better option is to have the control service consume its own events and define a rule that counts the rules that have been evaluated. Every time an `audit-success` or `audit-failure` event is received, the rule counts the correlated events and emits an `audit-completed` event when the expected number is reacted. But what happens when the expected count is never reached? Maybe the business process instance was never completed, and thus the audits of the final steps were never performed.

An easier example to reason about is knowing when a single rule has completed, such as the rule in the previous section that evaluates the combination of event types `A`, `B`, and `C`. Again, it is easy to know when the rule fails, but what if only two of the three expected events are ever received? This might be OK, or it might not be. Let's assume it is not OK and someone needs to be alerted when all three events did not occur. The obvious solution is to define another rule that asserts that all three events occurred, but the question becomes: *When do we evaluate this rule?*

The problem in both examples is that we are missing events but the rules are triggered by the presence of events. To solve this problem, we need to define a *time window* in which we expect the events to occur and set up a rule that asserts the presence of all the events when the time window expires. The following code block shows what the rules might look like:

```
const RULES = [{ id: 'w1', pattern: timer, eventType: ['A', 'B',
'C'], ttl: (uow) => uow.event.timestamp + 3600, // 1 hour },{ id:
'w2', pattern: expired, eventType: ['A', 'B', 'C'], expression:
(uow) => uow.correlated.filter((e) => ['A', 'B',
'C'].contains(e.type)).length < 3), message: 'ABC condition failed
with missing events', emit: auditFailed, }];
```

Rule `w1` creates up to three timers. When each of the event types is received, a timer is created—for example, when `A` is received, a window timer is created for `1` hour. When `ttl` expires, then rule `w2` checks to see if all three events are present and emits an `audit-failed` event if they are not. This ensures that there is coverage, no matter which of the events are received or not. For the business process audit example, there could be a single window based on the event that initiated the business process.

It can be a challenge to reason about missing events, but it is well worth the trade-off to be able to react to these complex conditions in real time. This simply highlights the benefits of using decision tables to model these rules and find all the permutations.

Now, let's look at a variation of the Control Service pattern that uses ML to look for actionable insights. This variation can be used as an alternative or a complement to CEP.

Leveraging machine learning (ML) for control flow

We have only scratched the surface of what can be accomplished with control services. Using rules to implement orchestration and CEP is clean and very powerful, but it is not the end of the road. We can certainly implement control flow with raw bespoke logic as well, but a very interesting approach that is emerging is the use of ML to steer control flow. For example, a control service could raise alerts and trigger business processes based on facial recognition or fraud and anomaly detection, or a control service could generate leads and personalized recommendations based on user activity.

To leverage ML, we need to look at both sides of the equation: **models** and **predictions**. Let's look at these in turn.

Models

When it comes to ML, it is all about the data. You need data, data, and more data. The more data you have, the more accurate your models will be. Conversely, if you do not have enough data, then you will most likely be better off using rules instead of ML.

Fortunately, we have the perfect source of data for **training** models—the **facts** in the *event lake*. This is one of the benefits of the *event-first* approach. The system is continuously emitting and collecting the facts we need for analytics in general and, more specifically, for training ML models.

You can use the data in the event lake to train and experiment with your initial models. If you do not have enough historical data, then you can start with rules to bridge the gap while you collect enough data. It's not unreasonable to expect that you will need a good year's worth of data or more. Once you have your models in place, you will periodically use the event lake to retrain your models. If you have enough traffic volume, you may even continuously retrain your models by consuming from the stream.

As you gain experience you will have ideas for better models. You can use the event lake to experiment with these hypotheses by **replaying** the events to validate the outcomes of the model against the historical results.

Predictions

A control service leverages ML to perform the **evaluation phase** of its processing logic. The objective is to *predict* whether an event represents

something interesting. If the event is of particular interest, an event is raised so that the system can **react** accordingly. The following diagram depicts the resources involved:

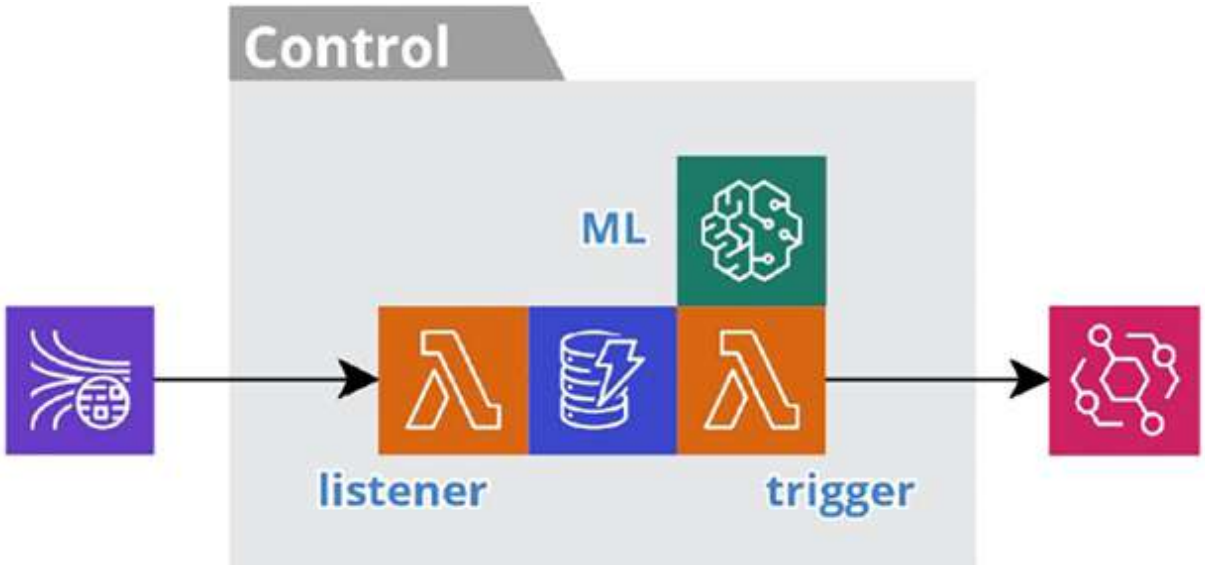


Figure 8.11: ML control service

The **listener** function collects events in a micro events store to provide idempotence. The **trigger** function invokes an **ML** service to *evaluate* the event. If the evaluation returns an *actionable* outcome, the **trigger** function then produces an event to initiate a business process, accumulate metrics, and more.

The possibilities for *evaluation* algorithms are endless—for example, a fraud-detection algorithm would return a score regarding the likelihood that an event was fraudulent, or a facial-recognition algorithm could determine whether or not a person of interest is present in an image associated with the event. This is an exciting arena where I expect to see a lot of experimentation and the discovery of new ways of evaluating events.

We have just scratched the surface of what is possible with control services. The *boundary* services simply provide *external* access to the system,

whereas the *control* services define the *core* of the system. As you experiment with control services, you will find more ways to control the flow of your system by reacting to *lower-order* events and generating *higher-order* events.

Summary

In this chapter, we dug into the details of the Control Service pattern, and you learned how to use it to implement inter-service collaboration such that changes in business policies have minimal impact on the boundary services.

We looked at variations on the pattern that handle orchestration and compensation transactions and you learned about the ACID 2.0 principles and how to use them to create event-sourcing snapshots. You also learned how we can use CEP and ML to initiate business processes.

In the next chapter, we will see how running in multiple regions optimizes responsiveness and resilience. We will cover regional routing, regional replication, regional health checks, and regional failover.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below: https://discord.gg/B8TubSxc35

9

Running in Multiple Regions

We covered the last of the three autonomous service patterns in *Chapter 8, Reacting to Events with More Events*. Throughout this book, we have focused on creating bulkheads within our systems to control the blast radius when things go wrong. Our autonomous services have inbound and outbound bulkheads to protect them from upstream and downstream services. Plus, we use the natural bulkheads of cloud accounts to protect autonomous subsystems from each other.

Now we turn our attention to multi-regional deployment. We will use cloud regions as another *bulkhead* to protect our systems from cloud provider disruptions. We will learn how to failover fast from an unhealthy region to a healthy region, so our systems continue running without downtime.

In this chapter, we're going to cover the following main topics:

- Justifying multi-regional deployment
- Choosing a regional topology
- Preparing for regional failover
- Checking regional health
- Configuring regional routing
- Replicating across regions
- Dissecting regional failover
- Addressing intersystem differences
- Implementing multi-regional cron jobs

Justifying multi-regional deployment

Multi-regional deployment is a hotly contested topic. There are many arguments for and against it. Which side of the argument you fall on depends on where you are standing. In other words, context matters. For example, it is more difficult to run traditional systems in multiple regions than it is with serverless systems. So, for our serverless systems, we are more inclined to favor the multi-regional approach.

Let’s look at why we should run our serverless systems in more than one region. The arguments revolve around risk, cost, and user satisfaction.

Regional disruptions

It is inevitable. Sooner or later a given cloud provider will experience a news-worthy regional disruption. It is not a matter of *if* but of *when*. In my experience, this happens about every 2 to 3 years or so. These major disruptions typically affect multiple cloud services and can last for hours, usually over five hours. Minor disruptions happen sporadically, impact fewer cloud services, and usually last less than a couple of hours. Either way, these disruptions can have a serious impact on the availability of our systems.

During a regional disruption, many systems have no recourse and they become unavailable because they only work within a single region. Meanwhile, other systems barely experience a blip in availability because they run in multiple regions. It only takes one bad experience to learn the value of multi-regional deployment.

I recall my experience during the AWS S3 disruption of 2017. Our system made heavy use of S3, and it was only running in the us-east-1 region where

the disruption occurred. Our monitoring system notified us when the disruption affected our site, but there was nothing we could do. There was nothing for us to fix and it would have taken too long to scramble and stand up everything in the us-west-2 region. All we could do was sit back and wait for the disruption to end and hope our users would forgive us.

Since then, AWS has added many features that enable our serverless systems to run in multiple regions. Throughout this chapter, you will learn how to use these features. With these features in place, a regional disruption should be a non-event. It should just be a blip on our observability dashboards.

Furthermore, once we have a system running in multiple regions, we can take advantage of these isolated environments to perform **regional canary deployments**. This technique helps teams mitigate the risk of a new deployment by smoke testing it in production in a single region before pushing it out to the other regions.

We will learn how to weave this technique into our CI/CD pipelines in *Chapter 11, Choreographing Deployment and Delivery*.

Nominal cost

A common argument against multi-regional deployment is that the cost and complexity are too high. However, this argument does not apply to serverless systems. First, serverless deployments are completely automated, which means adding regions is easy. We will see how to deploy to multiple regions in *Chapter 11, Choreographing Deployment and Delivery*.

Next, serverless systems scale to zero. The pricing is per transaction, so there is zero cost if there are zero transactions. This means that a secondary region in a primary/secondary configuration will not add cost when it is not

in use. Similarly, in an active/active configuration, we spread the transactions across the regions, so again there is no added cost for more regions.

We do incur costs for data replication in multiple regions. However, data replication, and its cost, are a necessary part of any disaster recovery strategy. Furthermore, the complexity of serverless data replication is significantly lower because its configuration is turnkey, and the cloud provider manages its operation. Therefore, the total cost of ownership for multi-regional serverless data is very reasonable.

Finally, serverless services handle the complexity of operating in multiple availability zones, and the cloud provider has included the cost in the price of the service. This frees us to refocus that development effort on developing the regional health checks for our multi-regional solution. We will cover this shortly in the *Checking regional health* section of this chapter.

Higher user satisfaction

The bottom line is that multi-regional deployment is in the best interest of our users. Our job is to deliver business value. Suppose users cannot use our system during a regional disruption. In that case, the price of having dissatisfied users may be significant if they can switch to a competitor or if they cannot complete their mission.

On the positive side, with an active/active configuration, users will experience better performance thanks to the reduced latency of accessing resources in the closest region. Plus, during a regional disruption, the users in the other regions will experience little to no impact. This last point is significant because the reasons for a regional disruption can be unpredictable and even unprecedented. Active/active ensures that some users are

unimpacted when we learn about a new regional disruption scenario the hard way.

Of course, some autonomous services are not critical and could run in a single region, but (as we will see shortly) the multi-regional settings are boilerplate, and it takes more effort to exclude them. Conversely, some services are very critical and require nuanced considerations when running at scale in multiple regions. Fortunately, as we will see in this chapter, we have already been preparing our serverless systems to run in multiple regions and the additional considerations are not daunting.

Now that we have justified multi-regional deployment, let's see how we can use regions in different topologies.

Choosing a regional topology

The terminology for regional topology is pretty cut and dried, but when we get down into the weeds it is common to find lots of variations in the overall interpretations and implementation details. In the context of serverless systems, we have two main options, *primary/hot-secondary* and *active/active*, with some variations. Let's look at these in the order we might adopt them.

Primary/hot-secondary

Primary/secondary is the most common regional topology for traditional systems. We route users to the primary region (east) and we failover to the secondary region (west). The failover process usually involves manual intervention, so it is more correct to call this a *primary/cold-secondary* topology.

For our serverless systems, we employ a primary/hot-secondary topology. We are deploying all our services in both regions, and they are ready to go because we are replicating the data in near real time, as we will see in the *Replicating across regions* section. We also automate regional failover using the regional health check service that we will cover in the *Checking regional health* section and the routing that we will cover in the *Configuring regional routing* section.

This is often a good starting position for new serverless systems because it is an incremental step towards a fully *active/active* topology. It may also be the best approach for systems that have external dependencies with limited support for multiple regions. We will look at some of these considerations in the *Addressing intersystem differences* section of this chapter.

Active/active

Active/active is the most complex regional topology. It is an incremental evolution of *Primary/Hot-Secondary*, but we need to carefully consider the concurrency characteristic of the various usage scenarios.

For example, if multiple regions update the same data concurrently then one region could overwrite the updates of another. These scenarios are usually in the minority, but we need to account for them nonetheless.

Let’s look at two variations of active/active: balanced users and balanced data processing.

Balanced users

Serverless systems need to support users in many geographic regions. Our food delivery system as an example runs nationwide and could expand into

other countries. We want all users to experience the best performance, so we balance the users across multiple regions using **latency-based routing**. This ensures that we will route users to the closest healthy regions. It also means that a regional disruption will not directly affect users in a health region when another region is unhealthy. Conversely, in a *Primary/secondary* topology, the disruption will affect all users.

In our example, there is no regional concurrency. A customer places an order for a specific restaurant location. If the user travels to another region and places an order, then it is a separate order in a separate region. The customer's order history replicates so it is viewable in all regions, but there is no contention. A traveler could schedule an order for delivery when they get home after their regional flight, but the time it takes to travel home is much longer than the replication latency. So again, there are no concurrency concerns.

An inventory system, on the other hand, is an example of a system that could experience contention as the inventory depletes. For example, a data model that relies on incremental updates, such as "update count by negative one where count is greater than zero," will experience contention if the transactions occur faster than the replication latency. In this case, we could keep the read model *active/active* and have the write model use *Primary/secondary*. The bottom line is that we need to pay attention to our write models in an active/active topology, which leads to the topic of *balanced data processing*.

Balanced data processing

When a user invokes a command on a BFF service, that action causes a **chain reaction** as downstream services react to the domain event. This data

processing occurs asynchronously in the same region as the initiating command and the results replicate in the other regions. In an *active/active* topology this results in a natural balancing of the background data processing based on the geographic location of the users.

However, not all data processing occurs as a direct result of a user action. For example, in the *Implementing multi-regional cron jobs* section, we will see how we can ensure that we only start scheduled jobs in a primary region to avoid duplicate data processing. In the *Addressing intersystem differences* section, we see that an ingress data feed from an external system may only support a Primary/Secondary configuration.

In most systems, it is perfectly fine to do this *background* data processing in a *primary/secondary* topology even if we configure our BFF services for an *active/active* topology. The results of this data processing will be replicated to the other regions in near real time, as per usual.

If your system has background scenarios like these, but the data is more time-sensitive, then you could implement content-based routing logic and dispatch the data to the best region to balance the data processing load. For example, if we are processing data for delivery schedules then we could route the data to a healthy region that is closest to the departure location of the shipment. This could minimize the latency for the users at those locations. In the *Replicating across regions* section, we will discuss why this approach is better than processing all events in all regions.

Now that we know our regional topology options, let’s review how we have already been preparing our serverless systems to run in multiple regions.

Preparing for regional failover

It may seem like a daunting task to design a system to run in multiple regions. This is understandable from a traditional, monolithic system point of view. Before the cloud, we did not always expect our systems to run in multiple data centers. We designed them to run in one on-premises data center at a time. The presentation tier assumed that a network connection would always be available. The data tier required strong consistency. Deployments were not automated, so failover was manual and usually involved a time-consuming restore of the database from the latest backup. Retrofitting these brittle systems to run in multiple cloud regions was painful. A Primary/Cold-Secondary topology was usually the best we could achieve.

In contrast, we have been designing our serverless systems to run in multiple regions all along. In today's highly connected world, the context and expectations for our systems are very different. We have reinvented our system architecture and these changes have prepared our systems to run in multiple regions as well. Let's review the significant advancements at the frontend and backend that have prepared our serverless systems to run in multiple regions.

Offline-first

In *Chapter 3, Taming the Presentation Tier,* we identified that our users are highly mobile and they expect their applications to work at all times. So, we must design the presentation tier of our serverless systems to withstand intermitted network connectivity and provide the user with a seamless

experience. As it turns out, this behavior is exactly what we need to ensure a smooth transition during a regional disruption. From the frontend perspective, there isn't any difference between a typical connectivity interruption and a regional disruption. We may even consider mobile connectivity to be a more significant problem since it may happen more often.

We follow an *offline-first* approach by fortifying the frontend with inbound and outbound caching (that is, bulkheads). When we read data, we employ a *stale-while-revalidate* caching strategy because it is better to provide users with stale data versus no data or errors. We proactively cache reference data and we refresh all queries in the background to ensure the most up-to-date information possible. When we write data, we implement *session consistency* by storing the user's changes locally, before sending them to the backend. If there is an interruption, we queue for the updates and seamlessly retry in the background. We also update the cache so that we can read the changes.

All the while, the frontend is transparent with the user about the state of the network connectivity and the freshness of the cached data. This provides the users with the situational awareness they need to make informed decisions during a disruption with the data they have at hand. Meanwhile, on the backend, our *eventual consistency* approach is helping to ensure a smooth transition as we fail over to a healthy region.

Protracted eventual consistency

Throughout this book, we have focused on creating autonomous services with inbound and outbound bulkheads to help ensure our serverless systems are robust and resilient. Upstream services produce their events even if

downstream consumers are unhealthy. Downstream services consume events and cache the data they need so they can still work even when upstream services are unhealthy.

Then we chain these services together following the **Command Publish Consume Query** (**CPCQ**) flow to choreograph and orchestrate complex business processes. Every event creates a **chain reaction** of downstream events that ensures the system is eventually consistent. When everything is going smoothly, we can expect consistency in *near real time*. When a service is unhealthy it may not consume events right away, but it will eventually consume all the required events when it recovers.

Again, this behavior is exactly what we need to ensure a smooth transition during a regional disruption. During a disruption, these chain reactions may *slow down* because of the increased error rate in the unhealthy region. But sooner or later the events will go through, and the updates will replicate across regions, as we will cover shortly in the *Replicating across regions* section. We can think of this as a period of **protracted eventual consistency**. It may take a little longer for the system to become consistent, but it will become consistent. Meanwhile, the transparency of the frontend keeps users apprised of the progress.

During this period, the increased error rate will result in more retries, but this is not a problem because we have built *idempotence* into every autonomous service. Transactions will trickle through since the error rate is usually not 100%. This ebb and flow will have an impact on the order in which we receive events. But again, this is not a problem because we employ techniques, such as *inverse oplock* and *event-state-transfer*, so that we are tolerant of receiving events out of order.

In other words, the *resilience* of *eventual consistency* shines through during a regional disruption. We will see shortly how we can improve on this with a global bus in the *Configuring regional routing* section. But all in all, we have already prepared our serverless systems to run in multiple regions.

Now let’s see how we can check the health of a cloud region.

Checking regional health

We need to check the health of the cloud regions we are using so that we can identify when it is time to fail over from an unhealthy region. Like so many other things, serverless is a bit different when it comes to health checks. At first glance, this may seem like a familiar exercise, but *regional health checks* are different from traditional health checks. They have a lot in common, but regional health checks work in a *higher order*.

Let’s review how traditional health checks work and then use that as a juxtaposition for our regional health checks.

Traditional health checks

In a traditional, non-serverless system we need to run clusters of machine instances across multiple availability zones, and we place a load balancer in front of each cluster to route traffic across the machines. Before a load balancer will route traffic to a given machine instance it needs to know whether the instance is healthy or not.

Each *machine* implements a health check endpoint that a load balancer pings regularly to check the health of the machine. These health checks test the health of the resources that the machine needs to use, such as CPU, memory, and network I/O. If any of the necessary resources are unusable then we flag

the machine instance as *unhealthy*. As a result, the load balancer will stop routing traffic to the unhealthy machine instance and *autoscaling* will replace the instance with a new instance.

Managing clusters, health checks, and autoscaling is tedious and error-prone. Fortunately, serverless services, such as AWS Lambda and AWS API Gateway, handle all this heavy lifting for us. **This allows us to refocus our efforts at the regional level.**

Regional health checks

Regional health checks work at a higher order than traditional health checks. Instead of routing traffic across multiple machines, we are **routing traffic across multiple regions**. Instead of checking the health of individual machines, we are **checking the health of whole regions**. Instead of checking the health of low-level resources, we are **checking the health of cloud services**.

That last point is where we see the biggest difference between traditional and regional health checks. The resources we need to test are entire cloud services versus low-level resources like memory and CPU. We need to test whole cloud services, like AWS Lambda and Amazon DynamoDB, to identify whether these services that we rely on are running properly in a given region.

If any one of these cloud services is unhealthy in a region, then we want to stop routing all traffic to that region. This is an important point. When a cloud service is unhealthy there is nothing, we can do about it. We cannot fix it or spin up a new instance of it. The “instances” of cloud services, so to speak, live at the regional level. If one stops working, we will have to use it in another region. But **we do not want transactions to crisscross regions**.

If any of the necessary resources are unusable then we flag the entire region as unhealthy.

The tricky part is figuring out when a given cloud service is healthy or not. For some cloud services, this is straightforward, but for others, it takes a little more elbow grease. We will use a combination of metrics and a regional health check service.

Metrics

In *Chapter 12, Optimizing Observability*, we will see that cloud services emit many useful metrics that give us a window to see how our systems are performing. We classify these as *work* and *resource* metrics, with subcategories like rate, latency, utilization, and errors. The **error metrics** are a good place to start for health checks.

Some cloud services emit error metrics that provide a good indication that the service is not healthy. For example, AWS API Gateway emits a `5XXError` metric, DynamoDB emits a `SystemErrors` metric, and S3 emits a `5xxErrors` metric. When we see an elevated number of these errors over a sustained period then it is likely that those cloud services are unhealthy.

The following `serverless.yml` fragment creates a `CloudWatch Alarm` that will alert us about a regional disruption:

```
Resources: RegionalHealthAlarm: Type:
AWS::CloudWatch::CompositeAlarm Properties: AlarmName: regional-
health-check AlarmRule: ALARM(ApigHealthAlarm) OR
ALARM(DynamoDBHealthAlarm)... ApigHealthAlarm: Type:
AWS::CloudWatch::Alarm Properties: ComparisonOperator:
GreaterThanThreshold Threshold: 0 EvaluationPeriods: 5 Period: 60
Statistic: Sum Unit: Count Namespace: AWS/ApiGateway MetricName:
5XXError Dimensions: - Name: ApiName Value: regional-health-check-
service DynamoDBHealthAlarm: ...
```

In this example we are checking for API Gateway and DynamoDB errors. At a minimum, we are checking these metrics for our `region-health-check` service that we will cover next. It is also a good idea to check these metrics for other high-value services with frequent invocations. We can add additional metrics as well, but this is a good place to start.

Our `Threshold` is zero, the `Period` is one minute and the `EvaluationPeriod` is five minutes. In other words, if the service has at least one error per minute for five consecutive minutes then we will consider the region unhealthy. During those five minutes, end users will experience an increased error rate and our *offline-first* capabilities will smooth things over while we determine if there is a disruption or not. It may be possible to tune the evaluation period down from five toward two minutes. However, reducing the *evaluation period* too low will increase the likelihood of false positives that would cause a region to fail over prematurely and then fail back quickly. This is where we can leverage all our *observability* metrics to help us dial these settings into their optimal values. *Chaos testing* will help us tune this as well.

Unfortunately, not all cloud services emit metrics that give us a clear sign that something is wrong. To fill this gap, we will implement a regional health check service that exercises those services.

Regional health check service

Similarly to a traditional health check, we need to implement a *regional* health check that we ping regularly to test the health of the region. This will be an independent service that we run in each region with its own resources. This service exercises all the cloud services that we use so that we can get a clear picture of their health.

The following diagram depicts a regional health check service:

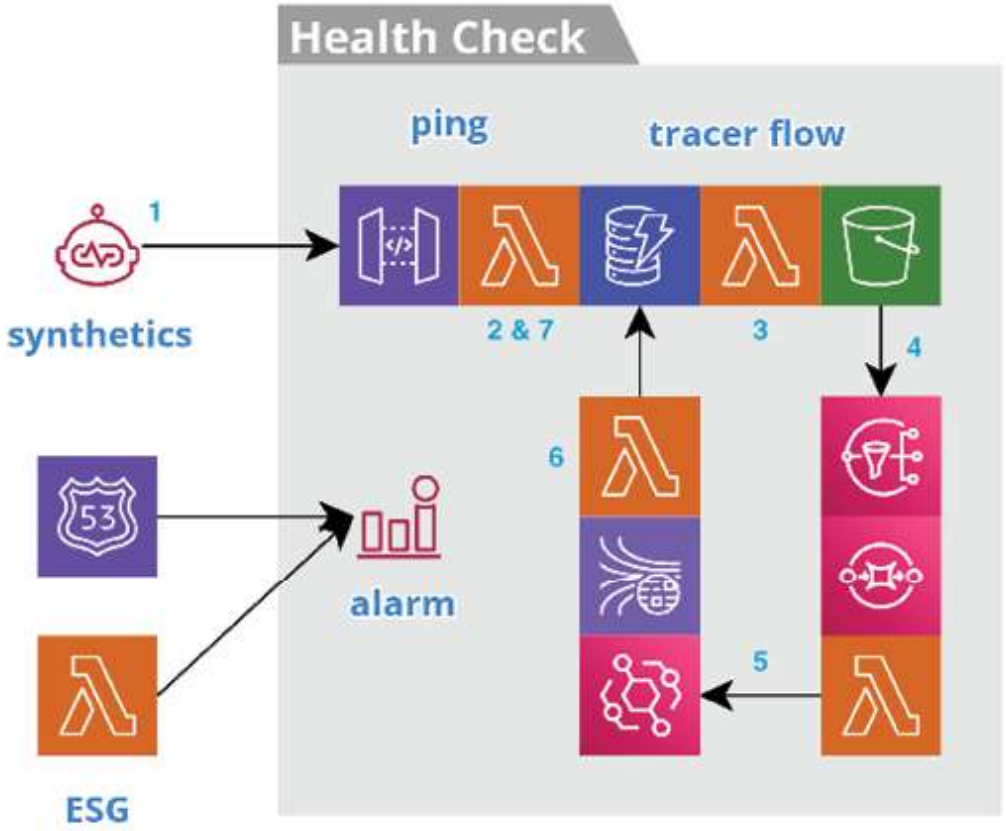


*Figure 9.1: Regional health check service The regional health check service asserts the health of the resources used by the autonomous subsystem by sending a **tracer** event through the services. If the tracer event does not flow through all the services before the next ping, then there is a good chance that one of those services is unhealthy. Here are the detailed steps as shown by the numbers in Figure 9.1:*

1. CloudWatch **Synthetics** calls the **Health Check** service once a minute through the **ping** endpoint implemented via API Gateway.
2. The **ping** Lambda function inserts a **tracer** record into the DynamoDB table.
3. The DynamoDB stream triggers a Lambda function to insert the tracer into an S3 bucket.

4. The S3 trigger sends the tracer to SNS, which sends it to SQS and onto a Lambda function.
5. The Lambda function publishes the tracer to EventBridge, which routes it to a Kinesis stream and onto a Lambda function.
6. The Lambda function updates the tracer in DynamoDB as Successful.
7. The next **ping** will return a 5xx error if the previous **tracer** was not updated as successful.

This **tracer flow** ensures that the metrics we are watching in the *Metrics* section have a steady stream of data to evaluate. Furthermore, we configure CloudWatch Synthetics to ping the opposite region, so that a disruption with the Synthetics service does not obscure other service disruptions.

Our CloudWatch composite alarm watches all the metrics and will change state after the evaluation period. From here, other services such as Route53 and our **External Service Gateway** (**ESG**`)` services will use the alarms to control their regional routing behavior.

The following `serverless.yml` fragment creates a `Route53 HealthCheck` that we will use to trigger regional failover:

```
Resources: RegionalHealthCheck: Type: AWS::Route53::HealthCheck
Properties: HealthCheckConfig: Type: CALCULATED HealthThreshold: 2
ChildHealthChecks: - Ref: ApigHealthCheck - Ref:
DynamoDBHealthCheck ApigHealthCheck: Type:
AWS::Route53::HealthCheck Properties: HealthCheckConfig: Type:
CLOUDWATCH_METRIC AlarmIdentifier: Name: Ref: ApigHealthAlarm
Region: ${opt:region} DynamoDBHealthCheck: ...
```

In this example, we create a Route53 health check for each of the metric alarms and we create a regional Route53 health check that aggregates the metric health checks so that our Route53 hosted zone will only need to test

the single regional health check. We will dive into the Route53 details in the *Configuring regional routing* section.

You can find a regional health check service template here: https://github.com/jgilbert01/templates/tree/master/template-regional-health-check

Next, we will see how to configure regional routing and use the regional health check.

Configuring regional routing

Now that we have selected a regional topology and we know how to check the health of our regions, we can set up our regional routing. We will start by setting up a **Content Delivery Network** (**CDN**) for our micro-frontend **single-page application** (**SPA**). Then we will configure our BFF services with regional and global endpoints, and then we will add these to the CDN. Finally, we will set up a global endpoint for our event bus.

Content Delivery Network (CDN)

A **CDN** is a globally distributed network of edge locations designed to deliver web content to users with low latency and high availability. A CDN, such as AWS CloudFront, is the perfect solution for delivering the static resources that make up a micro-frontend SPA, as we covered in *Chapter 3, Taming the Presentation Tier*.

During a regional disruption, we need to ensure that we continue to respond to requests for the application's resource bundles. The CDN is our first line of defense, as we will rely heavily on its cache to enable a smooth user experience as we fail over to a healthy region.

The following diagram depicts the CloudFront and S3 deployment for serving a micro-frontend application:

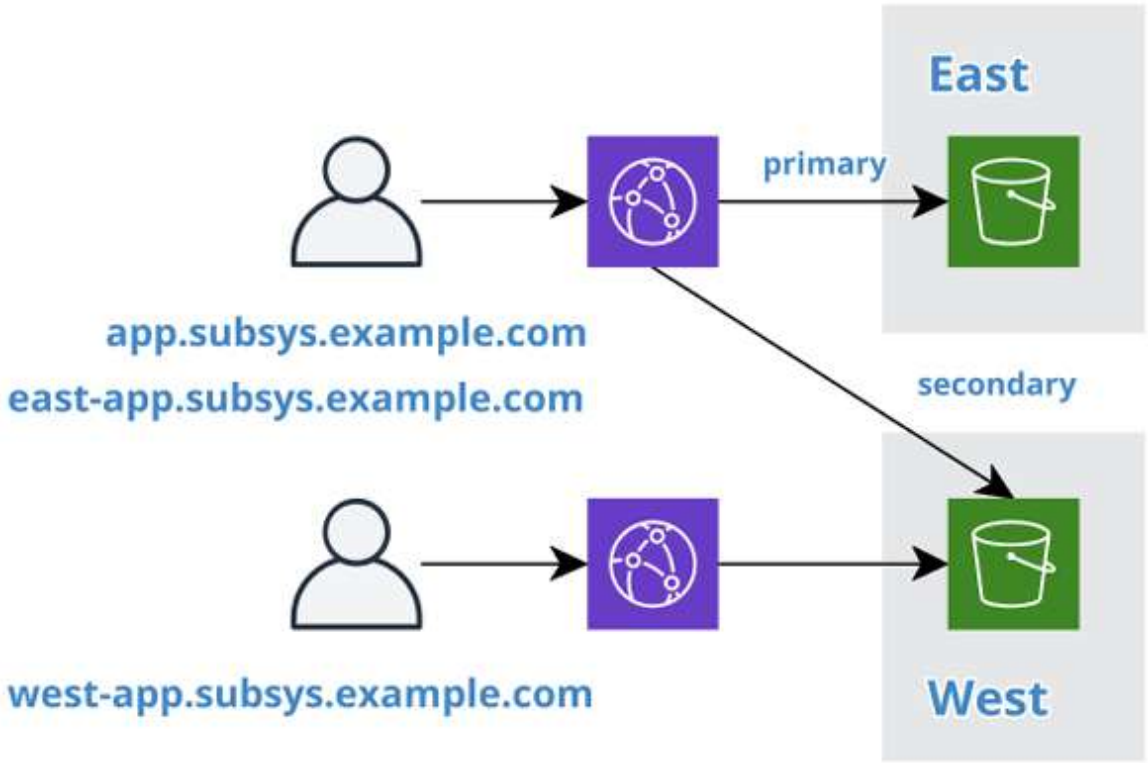


Figure 9.2: CDN

In each region, **East** and **West**, we create an S3 bucket where we deploy all the resources for a micro-frontend, such as HTML files, JavaScript bundles, and images. We deploy all the resources to both buckets so that we have an alternate if there is a regional disruption.

Next, we create a CloudFront distribution for our micro-frontend (`app.subsys.example.com`). We configure the distribution to use the bucket in the *eastern* region as the **primary** and the bucket in the *western* region as the **secondary**. We also set the `cache-control` headers on the resources so that we serve the resources from the edge cache as often as possible. As a result, during a regional disruption, the distribution will serve *cache hits* from the edge cache and *cache misses* from the secondary region.

We also set up a second distribution (`west-app.subsys.example.com`) and configure it to only use the bucket in the western region. We will use this domain name and distribution for our regional canary deployment smoke testing because we can be certain that it is accessing the latest deployment in the western region. We will cover regional canary deployment in *Chapter 11, Choreographing Deployment and Delivery*.

The following `serverless.yml` fragment creates a `CloudFront Distribution` with a primary and secondary origin:

```
GlobalDistribution: Type: AWS::CloudFront::Distribution
Properties: DistributionConfig: Origins: - Id: s3-primary
DomainName: ui-main.s3.us-east-1.amazonaws.com ... - Id: s3-
secondary DomainName: ui-main.s3.us-west-2.amazonaws.com ...
OriginGroups: Items: - Id: og1 Members: Items: - OriginId: s3-
primary - OriginId: s3-secondary FailoverCriteria: StatusCodes:
Items: [500, 502, 503, 504]
```

In this fragment, we create an origin that points to the eastern bucket and a second origin that points to the western bucket. Then we group these under `OriginGroups`. The group treats the first member as the primary and the other as the secondary. We also need to set up the `StatusCodes` that indicate when to fail over.

You can find a full CloudFront template here: https://github.com/jgilbert01/templates/tree/master/template-global-resources. This global resources stack owns resources used across all the services in an autonomous subsystem.

API Gateway

We covered the **Backend for Frontend** (**BFF**) pattern in *Chapter 6, A Best Friend for the Frontend*. These user-facing services support the micro-

frontend applications and the users who use them. We deploy BFF services and all their resources to multiple regions to provide users with a responsive and resilient experience, so we need a global endpoint that routes users to one or the other region.

Before we dig into global endpoints, let’s look at regional endpoints to highlight the basic building blocks.

Regional endpoints

We use regional endpoints when we need to send requests to a specific region. We will use the *western* endpoint for our *regional canary deployment* smoke testing to access the latest deployment. We will cover regional canary deployment in *Chapter 11, Choreographing Deployment and Delivery*. You will also use the regional endpoints during a regional disruption as you investigate the root cause and check the progress of the recovery.

The following diagram depicts a BFF service deployed to multiple regions with regional endpoints:

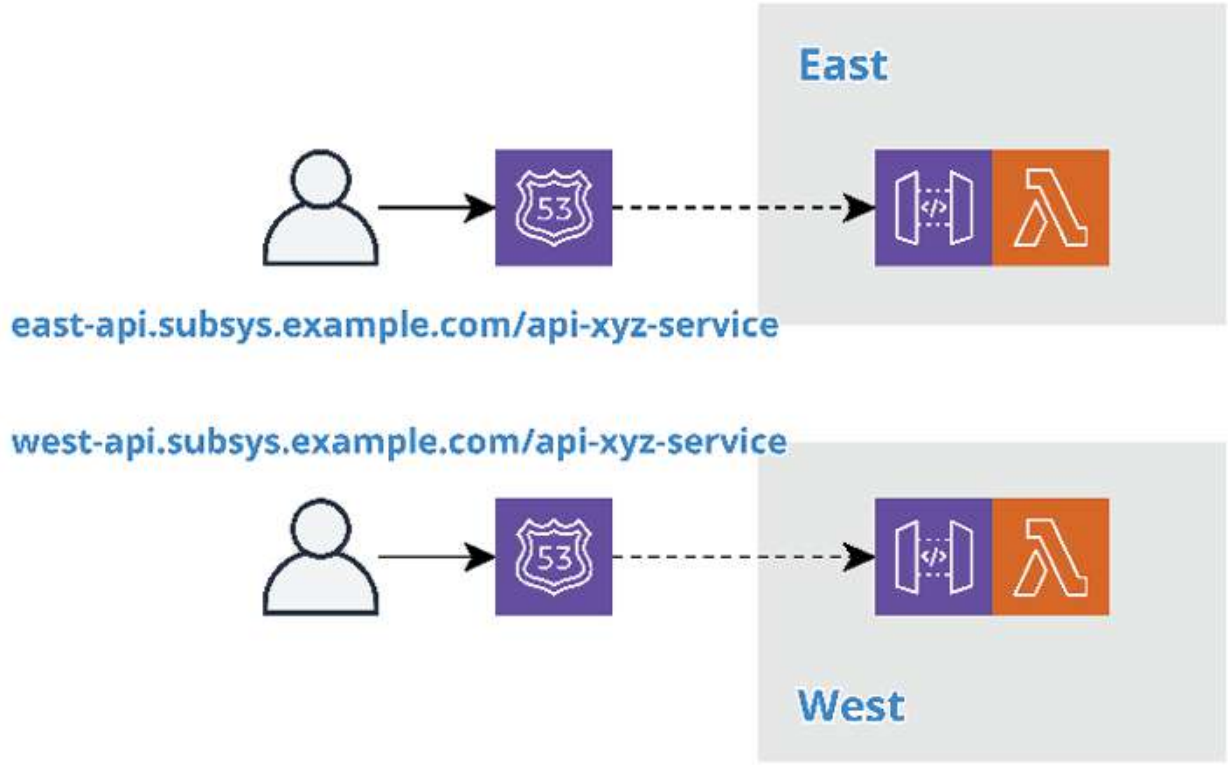


Figure 9.3: API Gateway regional endpoints First, we deploy custom domain names for each region: `east-api.subsys.example.com` *and* `west-api.subsys.example.com`*. Route53 will resolve these domain names to the correct regional instance of the API Gateway. Note that these domain names use an* `api` *prefix while the CDN domain names in the earlier section use an* `app` *prefix. This will be important when we combine the two in the next section.*

Next, we deploy the BFF service to both regions, **East** and **West**. Plus, we register each with a base path mapping of `/api-xyz-service`. This mapping tells the API Gateway service how to route to your specific service. We will also use the `api` prefix to distinguish BFF requests from micro-frontend requests.

The following `serverless.yml` fragment creates the regional endpoint for accessing BFF services in a given region:

```
ApiRegionalDomainName: Type: AWS::ApiGateway::DomainName
Properties: DomainName: west-api.subsys.example.com ...
ApiRegionalRecordSet: Type: AWS::Route53::RecordSet Properties:
HostedZoneId: ZZZZZZZZZZZZZZ Name: west-api.subsys.example.com.
```

```
AliasTarget: DNSName: Fn::GetAtt: [ApiRegionalDomainName,
RegionalDomainName] ...
```

In this fragment, we create an API Gateway `DomainName` for the given region. This registers our regional domain name with the API Gateway service, and it returns an internal endpoint that we can use in our DNS records.

Next, we create the DNS `RecordSet` and use the `AliasTarget` property to point our regional domain name to the internal endpoint returned by the API Gateway `DomainName`. Note that each autonomous subsystem has its own subdomain and corresponding Route 53 hosted zone.

You can find a full example here: https://github.com/jgilbert01/templates/tree/master/template-global-resources. This global resources stack owns resources used across all the services in an autonomous subsystem.

The following `serverless.yml` fragment registers a BFF service with the regional endpoint:

```
provider: endpointType: REGIONAL ... resources: Resources:
ApiRegionalBasePathMapping: Type: AWS::ApiGateway::BasePathMapping
Properties: BasePath: api-${self:service} DomainName: west-
api.subsys.example.com ...
```

In this fragment, we set the `endpointType` for the API Gateway to `REGIONAL` to disable the creation of a default CloudFront distribution. This will allow us to take full control of the CDN, as we will see shortly in the *CDN plus API Gateway* section. Then, we create a `BasePathMapping` to register the BFF service with the regional endpoint.

You can find a full example here: https://github.com/jgilbert01/templates/tree/master/template-bff-service.

Global endpoints

We use global endpoints to route requests to the best region based on our chosen topology and the health of the regions. For a *primary/hot-secondary* topology, it routes all users to one healthy region. For an *active/active* topology, it will route users to the closest healthy region based on their geographic location.

The following diagram depicts a BFF service deployed to multiple regions with global endpoints:

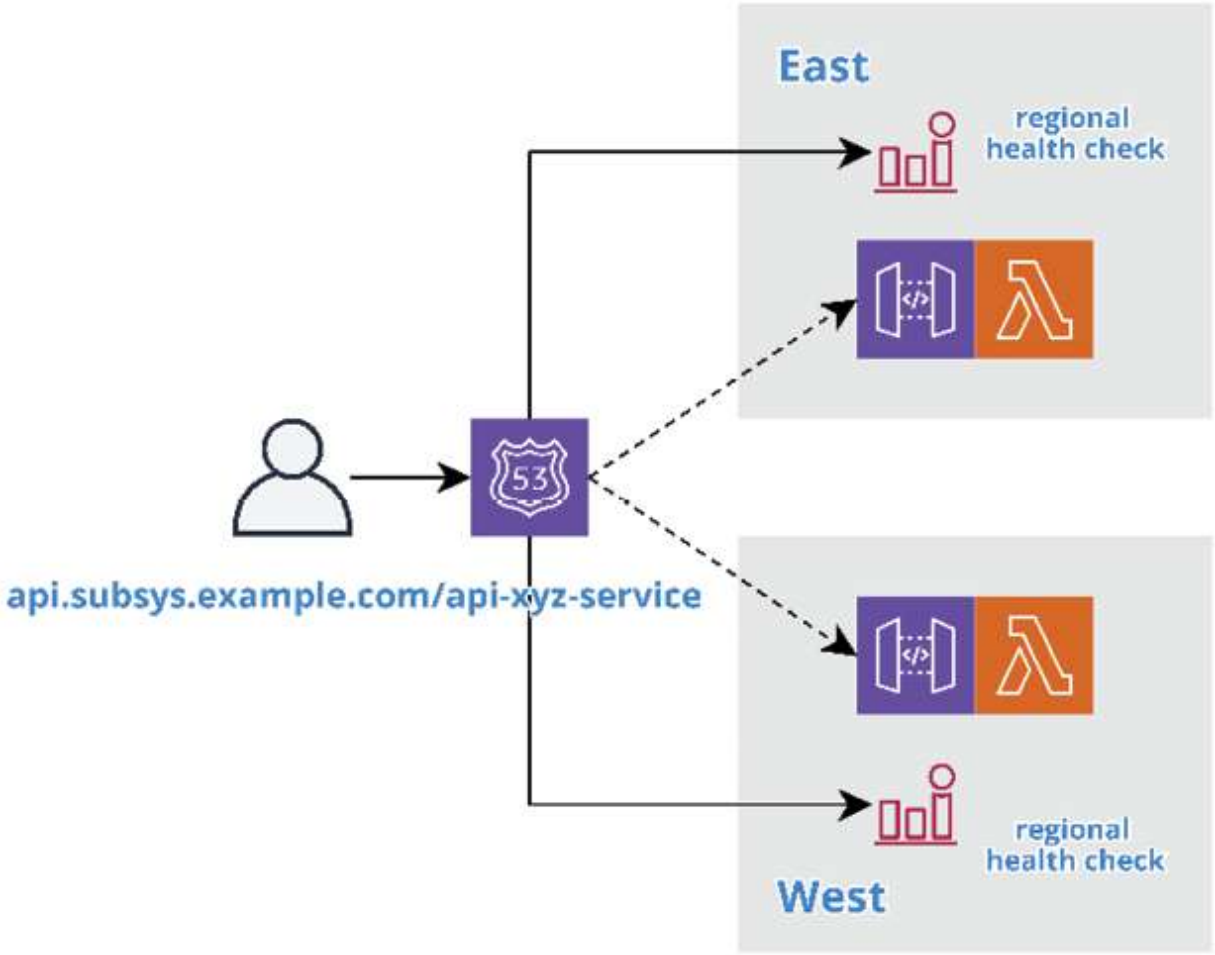


Figure 9.4: API Gateway global endpoints First, we deploy the global domain name: `api.subsys.example.com`*. When the frontend resolves the domain name, the cloud provider's* ***Domain Name System (DNS)*** *service, such as AWS Route 53, will resolve the domain name to the*

*best regional instance of the API Gateway based on our chosen topology. Route53 asserts the **health check** in each region before resolving to a region. It will not return the address of an unhealthy resource unless all the resources are unhealthy. The configuration is very similar to the regional endpoints we covered previously.*

The following `serverless.yml` fragment creates a regional `Route53 Health Check` that the DNS resolution will use:

```
RegionalHealthCheck: Type: AWS::Route53::HealthCheck Properties:
HealthCheckConfig: Type: CALCULATED ChildHealthChecks: - Ref:
ApigHealthCheck - Ref: DynamoDBHealthCheck
```

In this fragment, we create a `CALCULATED HealthCheck` that aggregates multiple `ChildHeathChecks`. These child health checks are based on the alarms we created in the *Checking regional health* section.

You can find a full regional health check template here: https://github.com/jgilbert01/templates/tree/master/template-regional-health-check.

The following `serverless.yml` fragment creates the global endpoint for accessing BFF services across regions:

```
ApiGlobalDomainName: Type: AWS::ApiGateway::DomainName Properties:
DomainName: api.subsys.example.com ... ApiGlobalRecordSet: Type:
AWS::Route53::RecordSet Properties: Name: api.subsys.example.com.
... HealthCheckId: 123456 SetIdentifier: ${opt:region} Region:
${opt:region} # latency-based # Failover: PRIMARY
```

In this fragment, we are setting up the DNS `RecordSet` for the global endpoint. Here, we focus on the differences from the regional endpoint configuration. First, we add the `HealthCheckId` property to enable failover. Next, we add the `SetIdentifier` and set it to the given region. This will make the record unique since we will have a record from each region with the same domain name. Finally, we set the `Region` property to enable a

latency-based routing policy for an **active/active** topology or we set the `Failover` property to enable a **primary/hot-secondary** topology. As you can see, starting with a primary/hot-secondary topology and switching to an active/active policy is a simple and incremental change.

You can find a full example here: https://github.com/jgilbert01/templates/tree/master/template-global-resources. This global-resources stack owns resources used across all the services in an autonomous subsystem.

CDN plus API Gateway

Having an API Gateway global endpoint is a minimum requirement for a multi-regional topology, but we can do better by combining the global endpoint with the CDN. First, it simplifies the frontend configuration because all BFF requests will be relative to the `app.subsys.example.com` domain name. This also removes the need for **Cross-Origin Resource Sharing** (**CORS**).

Next, it improves performance because CDNs, like AWS CloudFront, optimize the network traffic from the edge locations to the cloud regions. We can also use the `cache-control` headers on the BFF responses to cache results at the edge. This increases scalability and the performance for cache hits and reduces cost since it reduces the number of requests to the BFF serverless resources.

This last point is most important during a regional disruption because cache hits help smooth out the user experience as we fail over to a healthy region. In other words, cache hits minimize the number of times we route requests to an unhealthy region that would have returned a 5XX error response.

The following diagram depicts the BFF global endpoint deployed behind the CDN:

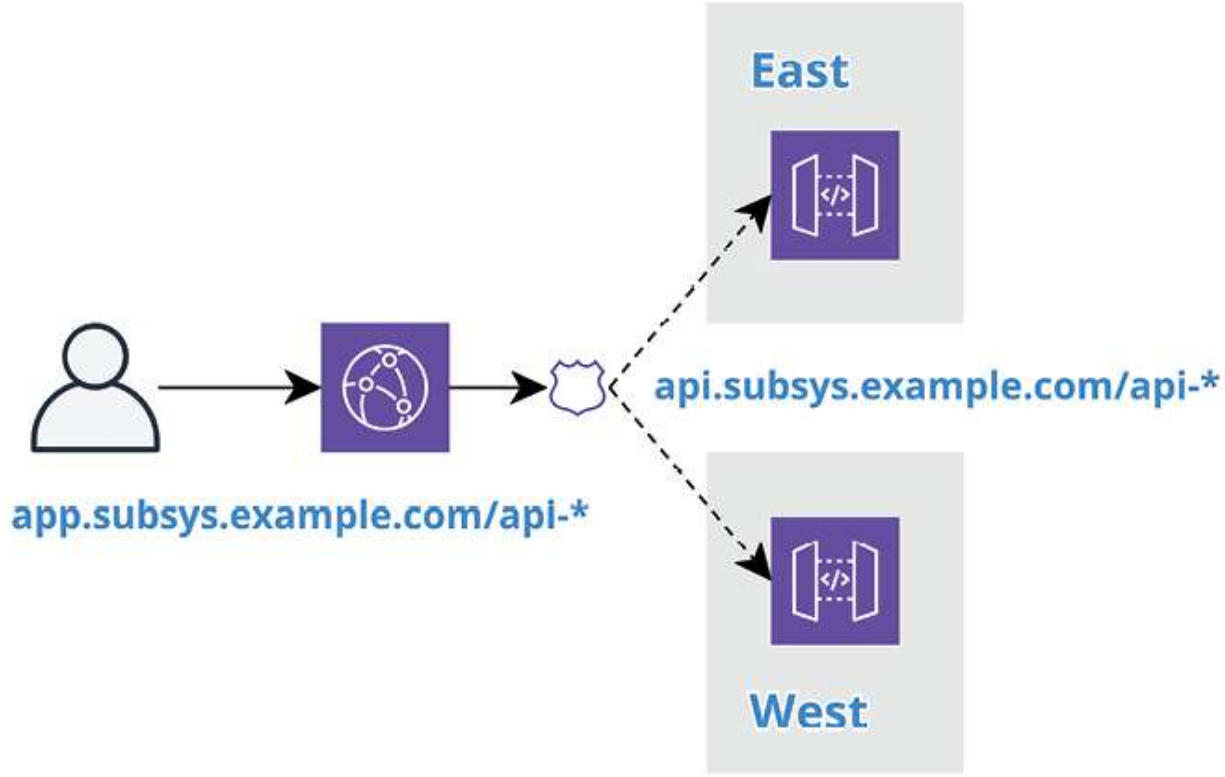


Figure 9.5: CDN plus API Gateway

In this configuration, we set up the CDN to forward all requests starting with `/api-*` to the API Gateway global endpoint. This is where the domain name prefixes (`app` vs. `api`) come into play. As we covered at the beginning of this section, CloudFront only supports *primary/secondary* failover, but we want to support an *active/active* topology for our BFF requests. The `api.subsys.example.com` allows us to forward these requests to the global endpoint that supports the topology we want to implement. This gives us the best of both worlds.

The following `serverless.yml` fragment extends the CDN example to include the api global endpoint:

```
GlobalDistribution: Type: AWS::CloudFront::Distribution
Properties: DistributionConfig: Origins: - Id: api DomainName:
api.subsys.example.com ... CacheBehaviors: - TargetOriginId: api
PathPattern: /api-* ...
```

In this fragment, we are adding the `api` global endpoint to the CDN as an origin. Under the `CacheBehaviors` property, we configure the CDN to source all requests that start with `/api-` from this origin.

You can find a full example here: https://github.com/jgilbert01/templates/tree/master/template-global-resources. This global-resources stack owns resources used across all the services in an autonomous subsystem.

Global bus

In the *Preparing for regional failover* section, we saw that a regional disruption can cause *protracted* eventual consistency. This delay is fine for most systems. However, if your system is more sensitive, then a global bus will allow **chain reactions** to fail over over to a healthy region and proceed at a normal pace.

The following diagram depicts a global bus using AWS EventBridge global endpoints:

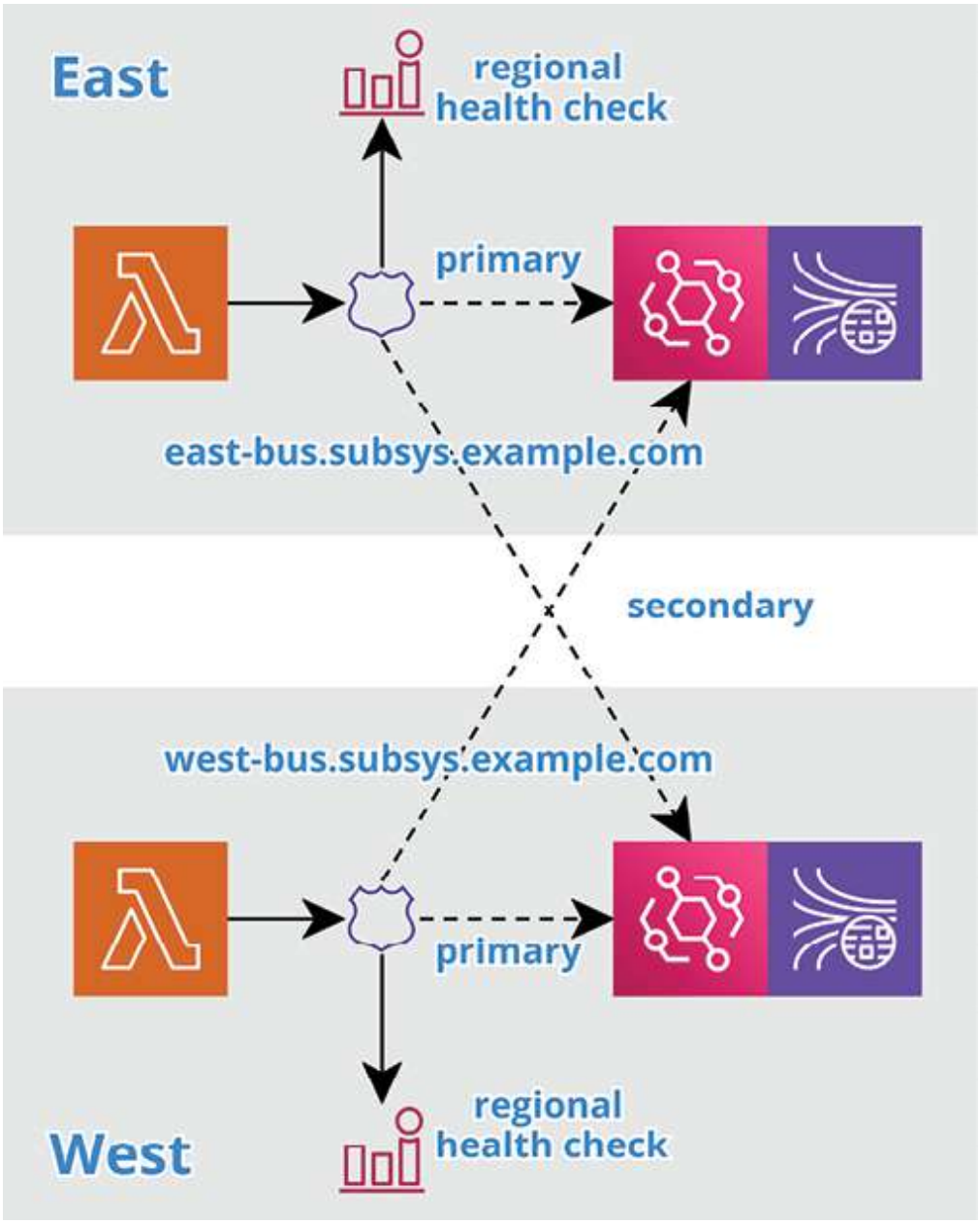


Figure 9.6: Global bus

In this example, we deploy an autonomous subsystem to the **East** and **West** regions per usual. We have a bus and channels in each region and producer functions publish events to the local (**primary**) bus. In addition, we have inserted a global endpoint *(east-bus/west-bus)* between the producers and the

bus, which is essentially a Route53 record set configured with a primary/secondary routing policy.

Under normal conditions, the routing policy will send requests to the local bus. This will ensure the lowest latency and minimize regional data transfer costs. However, we configure the global endpoint to use our *regional health check*, so that it can decide when to route requests to the **secondary** region. During a regional disruption, requests in the unhealthy region will fail over to the secondary and the *chain reaction* will continue in the healthy region.

The following `serverless.yml` fragment creates a global endpoint in the east with the west as the secondary:

```
Resources: Endpoint: Type: AWS::Events::Endpoint Properties: Name:
subsys-east-bus-endpoint RoutingConfig: FailoverConfig: Primary:
HealthCheck: arn:aws:route53:::healthcheck/123456 Secondary:
Route: west EventBuses: - EventBusArn:
arn:aws:events:east:xxx:event-bus/subsys-bus - EventBusArn:
arn:aws:events:west:xxx:event-bus/subsys-bus
```

In this fragment, the endpoint uses our regional health check to decide if the eastern region is healthy. If not then it will route the requests to the western bus.

You can find a full global bus template here: https://github.com/jgilbert01/templates/tree/master/template-event-hub.

Now, let's see how we can replicate data so that it is available across the regions.

Replicating across regions

In *Chapter 5, Turning the Cloud into the Database*, we covered the details of how an autonomous service leverages cloud-native datastores to create an inbound bulkhead to protect it from disruptions in upstream services. In this section, we will cover how regional data replication works at the change event level to protect autonomous services from regional disruptions. Specifically, we investigate multi-master and round-robin replication. But first, we need to address the advantages of change event versus domain event replication.

Change event vs domain event replication

Change events are emitted by a datastore's CDC stream as an autonomous service performs its fine-grained inserts, updates, and deletes. **Domain events** are emitted by an autonomous service to communicate to the system that an important course-grained state change has occurred. Autonomous services may publish change events to the bus to create a complete audit trail in the event lake, but they primarily use change events to decide when to publish domain events to the bus. These domain events trigger the chain reactions that we use to achieve the desired overarching business processes.

Some event bus services can replicate events across regions. We should avoid the temptation to use these features. **Replicating domain events across regions adds unnecessary cost and complexity.** It adds *cost* because we are performing **duplicate processing** of every domain event in each additional region. It adds *complexity* because we must reason about the consequences of all this duplicate work. There will be ever-so-slight variations in how each region processes the events. During a regional

disruption, the increased latency will accentuate these differences. In the worst case, each region could produce different results.

I recommend against replicating at the bus level. It has too many downsides, and it is too difficult to sort things out when a regional disruption goes bad. **It is better to replicate change events at the datastore level.** By all comparisons, protracted eventual consistency is much easier to reconcile when things go wrong. We can simply replay the events in the affected region and allow idempotence and order tolerance logic to sort things out.

Change event replication is much more straightforward than domain event replication. Each autonomous service reacts to domain events in a given region, performs its processing logic, and saves the results. Then we replicate the changes in the other regions. In other words, we perform the work once and replicate the output. Let's see how this works.

Multi-master replication

Running a database across multiple availability zones is already hard enough, so running across multiple regions may seem like a non-starter. Fortunately, fully managed, serverless datastores, such as DynamoDB and S3, implicitly run across availability zones and have turnkey support for multi-master replication across regions. This allows teams to focus their efforts on the business logic while implementing a multi-regional solution.

Figure 9.7 depicts the multi-regional deployment of a BFF service. The **East** region has an event hub with a **stream** and a **bus**, a BFF service with **command**, **listener**, and **trigger** functions, and a **datastore**. The **West** region contains an exact copy of all these resources, and we enable bi-directional replication between the datastores in both regions:

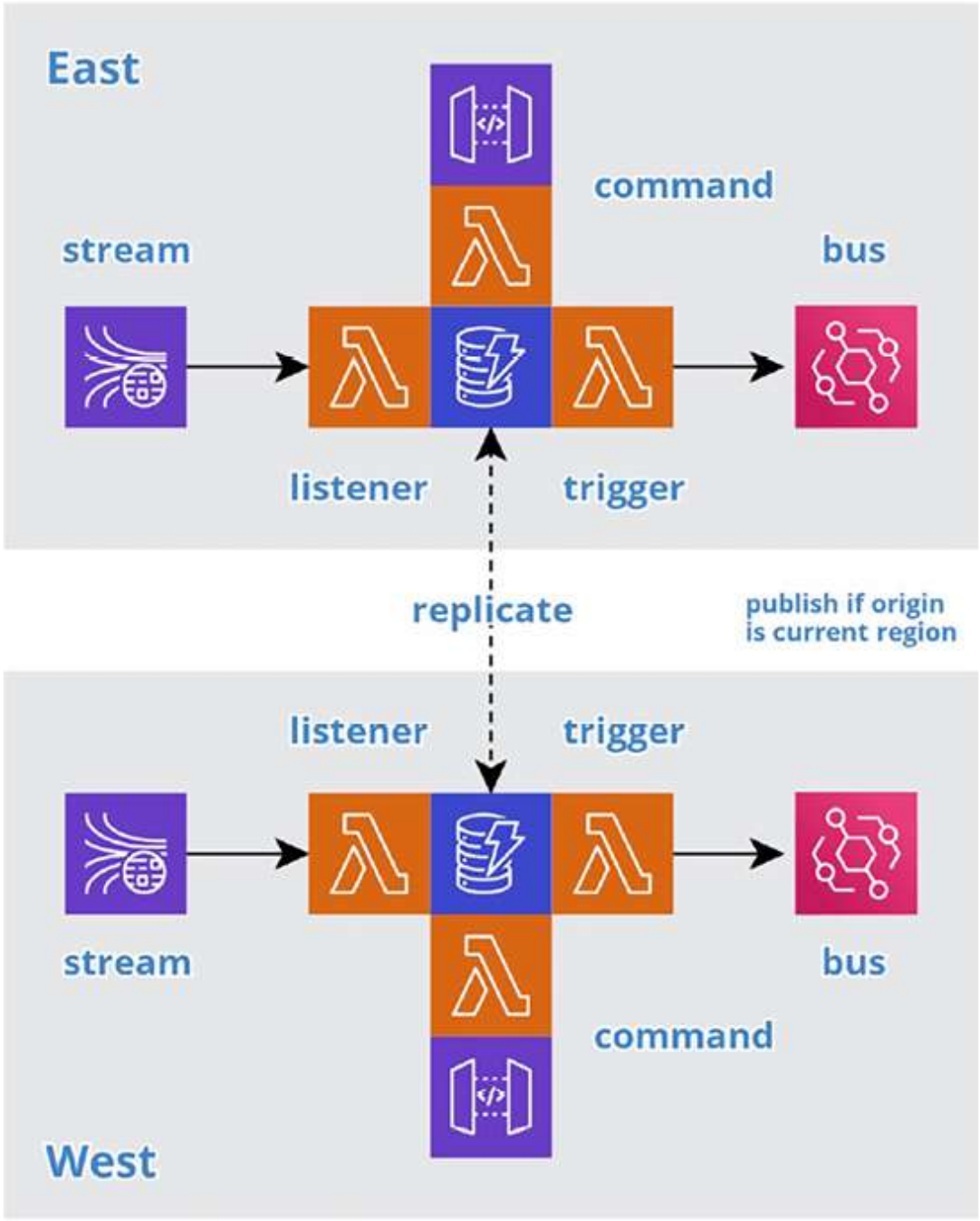


Figure 9.7: Multi-master replication

Under normal circumstances, with no regional disruption, an end user will access a system through the closest region, and the chain reaction of events caused by the user's actions is performed in the same region. For example,

when the **command** function executes in the **East** region, it writes data to the datastore in the **East** region. The **trigger** function then reacts to the change event in the same region, and it emits a domain event to the **bus** in the same region. All subscribers to the domain event, such as a **listener** function, will execute in the **East** region and materialize their view data in the **East** region. Meanwhile, all the data replicates to the **West** region to support the users in that region.

In the **West** region, after the data replicates, we do not want to invoke the **trigger** function and emit a duplicate domain event, because this will result in a duplicate downstream chain reaction that the system is already performing in the **East** region. Instead, we rely on the downstream services to replicate their own data.

This post-replication behavior will depend on the database service you are using. Some datastores do not emit change events to their CDC stream after replication, but some do. If so, then we can short-circuit the trigger function before it emits a duplicate event by comparing the current region to a region field in the record. If the datastore does not automatically add a region field, then we can add it and set it to the current region when writing the data.

The following `serverless.yml` fragment adds a regional filter to the trigger function on a DynamoDB stream:

```
functions: trigger: handler: src/trigger/index.handle events: -
stream: type: dynamodb arn: ... filterPatterns: - eventName:
[INSERT, MODIFY] dynamodb: NewImage: awsregion: S: [${opt:region}]
- eventName: [REMOVE] dynamodb: OldImage: awsregion: S:
[${opt:region}]
```

When we save a record, we set the `awsregion` field to the current region. Then we add the filter to the stream so that we only process the records that were inserted, modified, or removed by functions running in the current

region. This ensures that we do not duplicate the processing of any changes that originate from other regions.

It is important to understand how the multi-master replication solution of a specific datastore handles conflict resolution so that we can account for it in our designs. For example, DynamoDB follows a **last writer wins** reconciliation strategy. This is effectively the same as the **inverse-oplock** technique that we covered in *Chapter 5, Turning the Cloud into the Database*. However, DynamoDB reconciles at the record level, and we can apply the inverse-oplock technique at the field level. So, we would need to follow suit for any data with high regional contention. That said, we also want to avoid scenarios where multiple regions are contending for the same records, as that could result in replication thrashing and increase replication latency.

Round-robin replication

Not all databases support turnkey regional replication, yet we still need to replicate their data across regions to improve latency and support regional failover. These will typically be read-only datastores, such as Elasticsearch, that do not have a CDC stream. We also want to follow a consistent pattern of relying on services to replicate their own data. To accomplish this, we can add round-robin replication to the inbound flow of these downstream services.

Figure 9.8 depicts the multi-regional deployment of a BFF service that uses Elasticsearch. The **East** region has an event hub with a **stream**, and the BFF service has a **listener** function to consume domain events that contain data that we need to index in the **view**. Between the **listener** function and the **view**, we add a **topic**, such as Amazon SNS, a **replicator** function to

implement the round-robin replication, and a **save** function to populate the **view**. The **West** region contains an exact copy of all these resources:

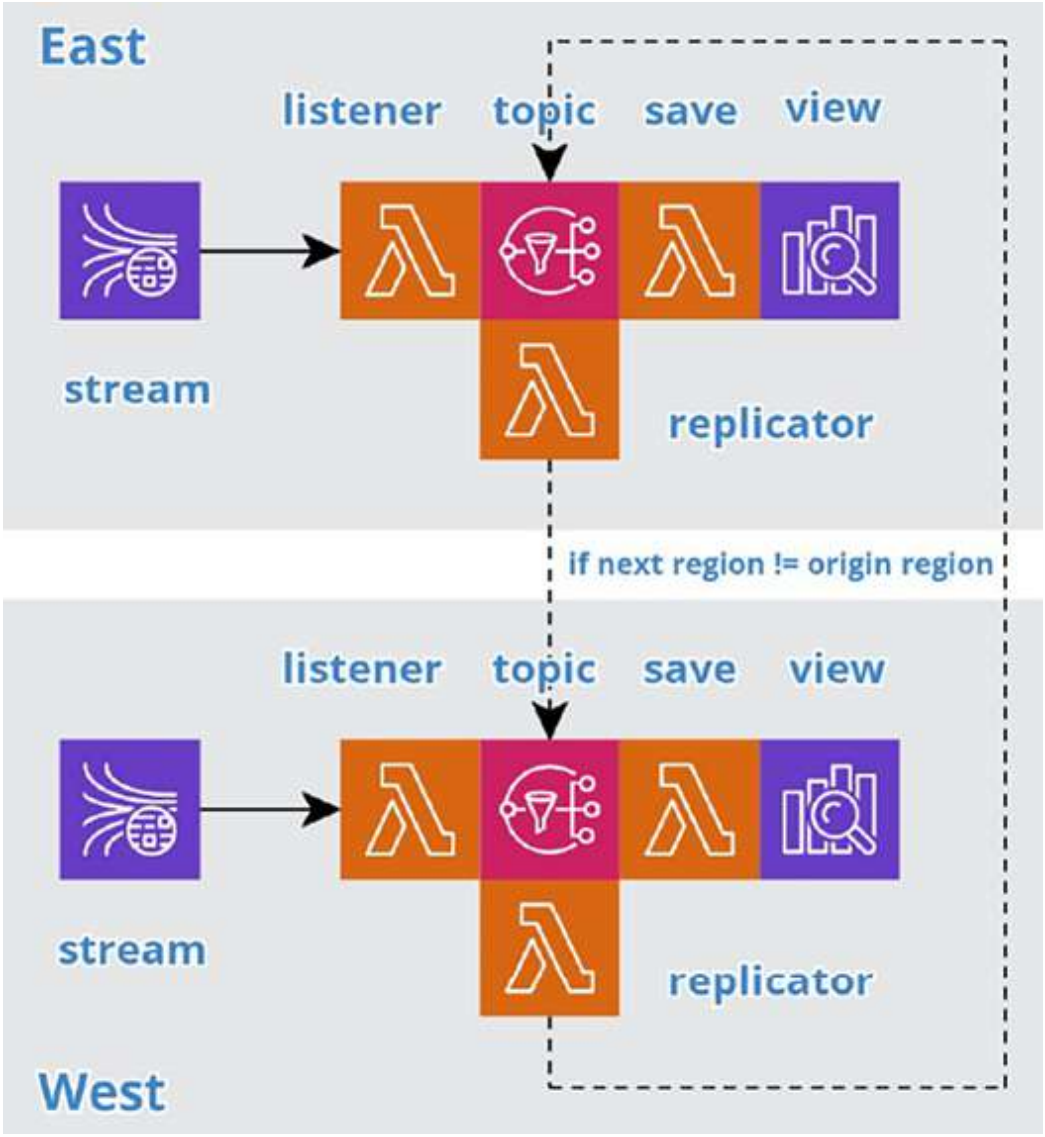


Figure 9.8: Round-robin replication

The **listener** function in the **East** region consumes domain events from the **stream** in the **East** region. It transforms the data to the desired format and sends a message to the **topic** in the same region, along with a message attribute that indicates the originating region. We perform the semantic

transformation in the **listener** function to ensure that we only perform this logic once, instead of per region.

The **save** function in the **East** region reacts to the **topic** in the **East** region and updates the **view** in the same region. Meanwhile, the **replicator** function in the **East** region also reacts to the same **topic**. The **replicator** forwards the message to the **topic** in the next region, as specified by an environment variable. In this example, we configure **East** to forward to **West** and vice versa.

The **save** function in the **West** region reacts to the **topic** in the **West** region and updates the **view** in the same region. The **replicator** function in the **West** region also reacts to the same **topic** and looks to forward the message to the next region. This flow will round-robin for as many regions as necessary, until the next region is equal to the originating region.

In this example, we have two regions—East and West. The data originates in the East region, so the East **replicator** forwards the data to the West **topic**. The West **replicator** does not forward the data to the East **topic** because East is the origin. If the data originates in West, then it will forward it to East, but not back to West.

This round-robin replication technique is easy to implement and maintain. For brevity, I did not include queues, such as Amazon SQS, between the topic and the functions, but this is a best practice to add more resilience. With SQS in place, we could have all the queues subscribe to all the topics, but this would result in N x N subscriptions, with N being the number of regions. This quickly becomes difficult to manage and requires multiple deployment passes to wire everything together, so it is not the preferred approach.

Now that we have covered all the multi-regional building blocks, let’s see how this all comes together to produce a smooth user experience during a regional disruption.

Dissecting regional failover

Up to this point, we have covered all the building blocks for running serverless systems in multiple regions, including regional health checks, regional routing, and data replication. Now let’s see how all this comes together during a regional disruption.

It is important to understand the various failure points and how the system should react as it fails over to a healthy region. Fortunately, our well-defined patterns provide us with a bounded set of function categories that we need to dissect. These are the query, command, trigger, and listener functions we have across the patterns. We will use these to frame this topic.

Query failover

Data is essential to the operation of any application. Cached data, even if it is stale, is better than no data, provided the application is transparent with the users about the age of the data and the status of the system. So, when an application performs a query, we prefer to return stale data over no data or errors. The **stale-while-revalidate** caching strategy is a key technique we employ for a smooth user experience during a regional failover.

The following diagram depicts the layers of cache and components that a query request passes through:

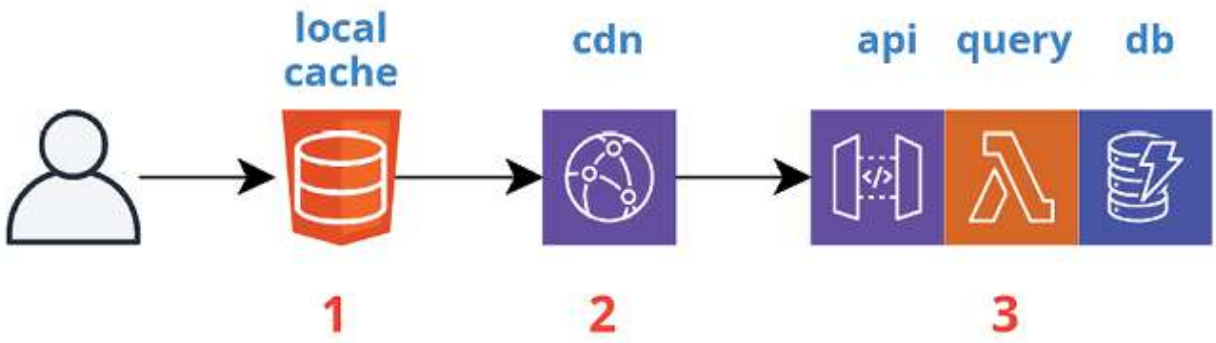


Figure 9.9: Query failover

We should use this flow for every query performed in an application, with different settings based on how quickly the data changes. During a regional disruption, the flow may short-circuit at different layers shown by the numbers in *Figure 9.9*:

1. The application will always look in the local cache first and return the data that is available. This may be the in-memory cache provided by the application and/or the browser cache managed by a service worker. If the data is stale, then we will retrieve the latest data in the background and refresh the cache. However, during a regional disruption, we will experience a higher error rate, so the application will continue to use the stale data to help ensure a smooth experience.
2. If the application does not have the data in its local cache, then the CDN may return the data based on the `cache-control` header settings. We should set this to a long period for slow-changing reference data so that it will be more likely to produce a cache hit. For fast-changing transactional data, we can set the header to a short value, such as five seconds, to increase the hit rate and help reduce the error rate during a regional disruption. For example, `cache-control: max-age=5, stale-while-revalidate=300`.

3. If the data is not in the CDN cache, then the request will go to the BFF service, which is also a cache of upstream data. The BFF service is the main point of regional failure in the query flow. The API gateway service, the function service, and/or the database service could be unhealthy and result in the BFF service returning a 5XX error to the application. In this case, the application should handle the error gracefully and periodically retry the query until we fail over to a health region with replicated data. This could take upward of five minutes, which is the health check's evaluation period.

Once we fail over to a healthy region, the user will have access to the data that has been replicated. However, when there is a regional failure, we should expect that replication latency increases as well. Therefore, we should compare the data with the local cache and only update the cache with newer data.

Command failover

Users also need to perform commands that act on the data in the system. During a regional disruption, these commands will produce 5XX errors. However, we do not want to burden the users with waiting and retrying the commands. The following diagram depicts the command flow with automatic retries:

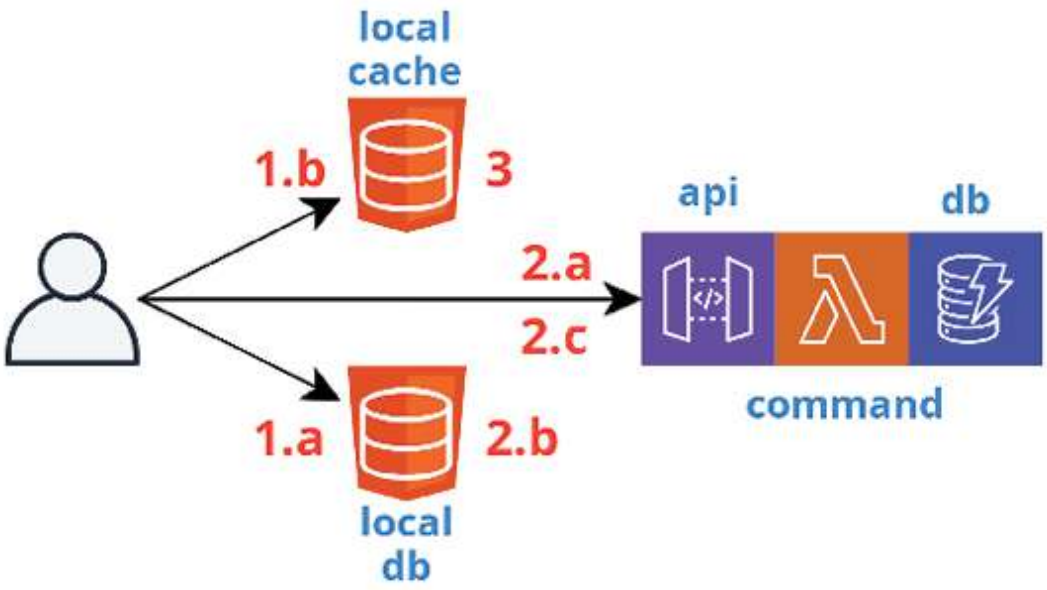


Figure 9.10: Command failover

In *Chapter 3, Taming the Presentation Tier*, we saw how *session consistency* and *offline-first* mechanisms allow the presentation tier to continue working even when the application has lost its connection to the network. As it turns out, this behavior is exactly what we need to ensure a seamless transition during a regional disruption as well. The numbers in *Figure 9.10* denote the order of flow:

1. First, the user enters data into a form and submits the command:
    a. As the user enters data, we store it locally in the browser's IndexedDB. This will ensure that we do not lose the user's work in progress during a disruption.
    b. When the user submits the command, we proactively update the local cache so that the application can read the data even if there is a disruption. We refer to this as **session consistency**. We should set the status of the data to **PENDING**, since the command has not gone through yet, and make this visible to the user for transparency.

2. Then we send the command to the BFF service to save the data and trigger the desired downstream reaction:
    a. As with the query flow, the BFF service is the main point of regional failure in the command flow. The API gateway service, the function service, and/or the database service could be unhealthy and result in the BFF service returning a `5XX` error to the application.
    b. When we receive a `5XX` response the `ServiceWorker` stores the command in an outbox table in the browser's IndexedDB to ensure that we do not lose the request.
    c. In the background, the ServiceWork periodically retries the queued command until we fail over to a health region and the BFF executes it successfully. This could take upward of five minutes, which is the health check's evaluation period.
3. Finally, we update the status of the data in the local cache.

Once we fail over to a healthy region, the user's commands will execute in the new region, the chain reaction of events will process in the new region, and the results will eventually replicate back to the failed region.

Trigger failure points

The *trigger* flow of an autonomous service comes into play after the *command* or *listener* function of the service has successfully persisted data to its datastore. The following diagram depicts the serverless services that play a part in the flow of a *trigger* function:

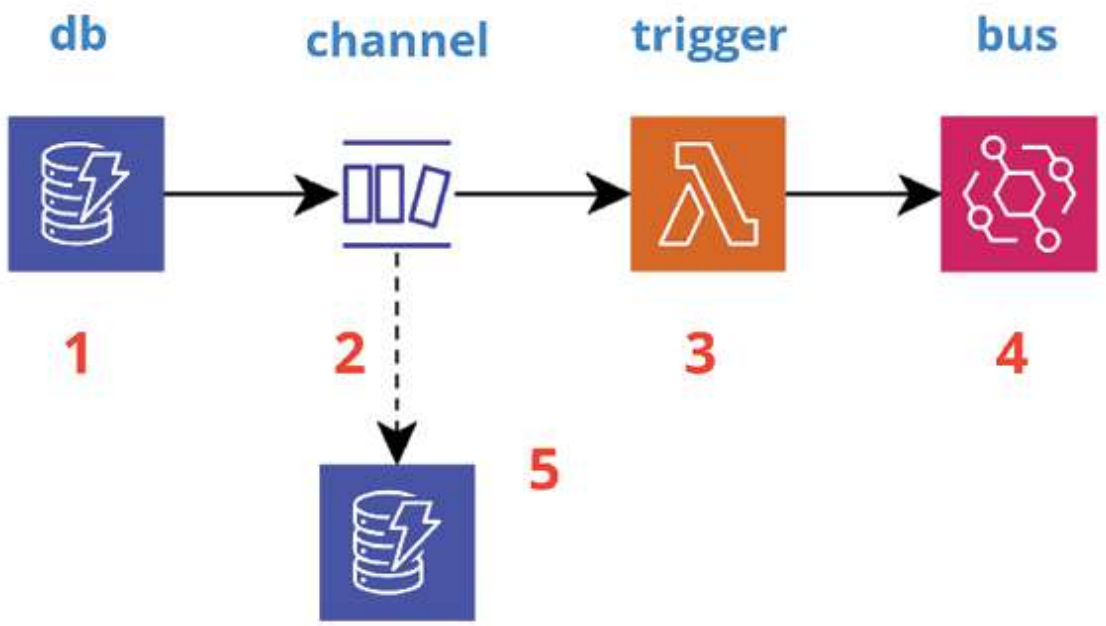


Figure 9.11: Trigger failure points

This flow starts after we have successfully stored data and the database has a durable entry in its transaction log. Each serverless service involved in the trigger flow is a potential failure point, as shown by the numbers in *Figure 9.11*:

1. The database service, such as DynamoDB, could be having trouble internally getting the change events to the channel.
2. The channel service, such as DynamoDB Streams, could be having trouble with writes and/or reads.
3. The function service, such as AWS Lambda, could be having trouble polling channels or invoking functions.
4. The bus service, such as AWS EventBridge, could be having trouble receiving events.
5. The database service could be having trouble replicating to other regions.

In most of these cases, there is little we can do other than rely on the protracted eventual consistency that we have prepared the system to handle. Our health check service will kick in and we will direct future transactions to a healthy region. The in-flight transactions are durable, so the unhealthy service will drain the channel in fits and spurts as it recovers.

The interaction between the trigger function and the bus is the one place we have some control. We can use a global bus, as we covered in the *Configuring regional routing* section, to send the events to a healthy region, where the chain reaction can continue at a normal pace. However, there will be a period of increased errors until the health check kicks in. So, we may need to resubmit some fault events and watch any **Dead Letter Queues** (**DLQs**).

Listener failure points

The *listener* flow of an autonomous service comes into play after the *trigger* function of an upstream service has successfully published an event to the bus. The following diagram depicts the serverless services that play a part in the flow of a *listener* function:

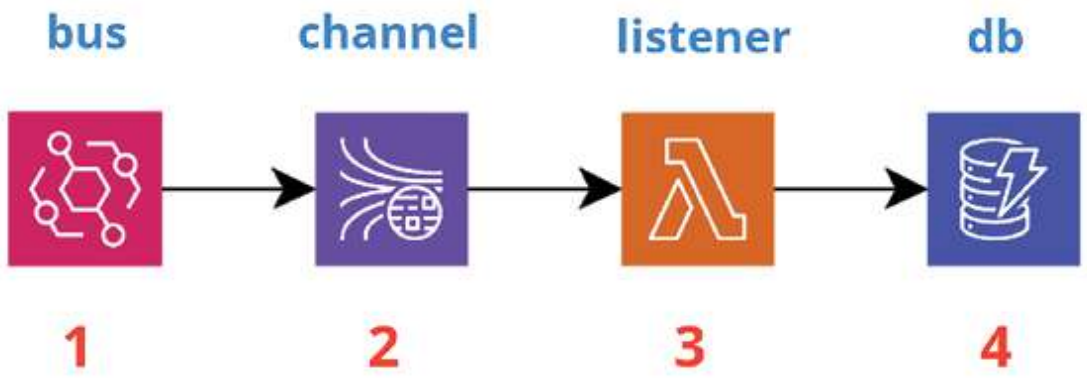


Figure 9.12: Listener failure points

This flow starts after the **bus** has successfully received an upstream event. Each serverless service involved in the listener flow is a potential failure point, as shown by the numbers in *Figure 9.12*:

1. The bus service, such as AWS EventBridge, could be having trouble internally getting events to the channel.
2. The channel service, such as AWS Kinesis, could be having trouble with writes and/or reads.
3. The function service, such as AWS Lambda, could be having trouble polling channels or invoking functions.
4. The database service, such as AWS DynamoDB, could be having trouble writing data.

Just like the trigger flow, in most of these cases there is little we can do other than rely on the protracted eventual consistency that we have prepared the system to handle. Our health check service will kick in and we will direct future transactions to a healthy region. The in-flight transactions are durable, so the unhealthy service will drain the channel in fits and spurts as it recovers.

The interaction between the listener function and the database is the one place we have some control. If the database is unhealthy then the iterator age metric for the listener function will spike as it enters a retry loop. These retries will allow the function to self-heal as the database recovers. Or we can proactively raise faults and resubmit them in a health region.

These scenarios are a great place for you to start dissecting the behavior of your system during a regional disruption. The nature of your data will dictate whether protracted eventual consistency is sufficient or if you need to take further action. Chaos testing and a monthly failover drill will help you uncover the nuances of your system and improve the failover process. The

observability metrics, we will be covering in *Chapter 12*, *Optimizing Observability*, will be invaluable during a regional disruption.

Now let's look at how we can enhance our integration to external systems when those systems follow a different multi-regional strategy.

Addressing intersystem differences

We covered the **External Service Gateway** (**ESG**) pattern in *Chapter 7*, *Bridging Intersystem Gaps*. We deploy ESG services to multiple regions just like other autonomous services, but with one major difference: ESGs interact with external systems that may have different architectures with different regional approaches. The purpose of an ESG is to encapsulate these differences so that they do not corrupt the rest of the system. However, addressing multi-regional differences is a multi-dimensional problem that can make your head spin as you consider the challenges from all angles. These challenges include the following:

- There is the perspective from your system and from the external system.
- Is the flow egress or ingress from your system's perspective?
- Will your system be using a push or pull technique?
- Is the regional disruption on your side or their side or both?
- Is the external system in the same cloud, a different cloud, or on-premises?
- Is its topology primary/secondary or active/active and what is the failover period?

- Does the external system supply a global endpoint, or will you have to perform routing on your side?

Let’s use the last point to organize this topic. We will consider whether the external system has a **global** or **regional** endpoint and look at those endpoints from the **egress** and **ingress** points of view. Along the way, we will see how the other dimensions play out.

External global endpoints

Let’s start with the happy path of an egress flow. A logical transaction starts by routing a request from a user of your system to the closest healthy region. From that point on, the chain reaction of events happens in that region. When we reach an ESG, we prefer it to invoke the external system in its closest healthy region. In the best case, the external system will provide global endpoints with automated failover, as we covered in the *Configuring regional routing* section. This will free the ESG from the heavy lifting and complexity of this responsibility.

Third-party SaaS systems will typically handle their regional routing and failover for you. Some enterprise-grade messaging systems have global endpoints as well. This means that your ESGs can push domain events to these external systems without regional concerns. However, you should still understand the details of their approaches. For example, if an external system is active/passive, then that means that some of your regional users will experience more latency than others. If the external system uses the same cloud provider as you, then it may be susceptible to the same regional disruptions.

When we experience a regional disruption on our side, we stop routing new transactions to the unhealthy region and the ESGs in the unhealthy region

will **drain** all those transactions that are already in flight from their channels and process them as quickly as possible. Meanwhile, the ESGs in the healthy region will send new transactions to the external system's global endpoint as well, so we need to confirm that the logic of the external system is **order tolerant**.

In *Chapter 7, Bridging Intersystem Gaps*, we covered how you can provide an **external API** that allows external systems to call your system to pull data, push events, and invoke commands. We also covered how your system can subscribe to the **webhooks** of an external system to receive events. We implement these endpoints using an API gateway and follow the same routing and failover approach outlined in the *Configuring regional routing* section. This makes the routing and failover of your system transparent to the external systems, just like we want external systems to make routing and failover transparent to your ESG services.

Legacy regional endpoints

Most legacy systems, particularly those running on-premises, only supply regional endpoints running in a primary/secondary topology and they assume that the calling system (that is, your system) will perform the regional routing. Worse yet, the failover process may be extremely protracted. Once again, the purpose of the ESG pattern is to encapsulate and hide these details.

An ESG may use these regional endpoints in an **egress** flow to push events to the external system or it may use them in an **ingress** flow to poll for changes and pull events from the external system. In either case, the **Circuit Breaker** pattern we discussed in *Chapter 7, Bridging Intersystem Gaps*, is your best choice for handling routing and failover. During the external

system's failover period, the circuit would stay **open** until the external team brings their secondary region online.

To handle routing, we can extend the Circuit Breaker pattern by adding an ordered list of regional endpoints. The ESG gives each endpoint its own circuit and only moves down the list once it has opened the preceding circuit. If the legacy system has multiple active sites, then the ESG can try each endpoint before opening the circuit or it can invoke the endpoints in a *round-robin* fashion to balance the load. Finally, if the legacy system supplies a health check, then we can open and close the circuits proactively.

Our **ingress** flows may require special handling because we may not require both of our regions to poll the external system. In this case, we can treat one region as the primary and the other region as the secondary. The ESGs will test the `RegionalHealthAlarm` that we created in the *Checking regional health* section. When we experience a regional disruption, then one region will stop polling and the other will start polling. To account for any delays in the chain reaction and replication from the unhealthy region, we can rewind the polling by 30 minutes or so to fill in any gaps and rely on idempotence to weed out any duplicates. When both regions are healthy, the ESG can balance the data processing load by performing intelligent content-based routing to send some events to the other region.

Now, let's look at how we can coordinate scheduled jobs across multiple regions.

Implementing multi-regional cron jobs

It may seem ironic, but scheduled jobs still play a role in event-driven systems, even though we strive to avoid the long latency of a batch cycle. For example, legacy integrations may need scheduled jobs and our **ESG services** can encapsulate those requirements. Plus, there are still legitimate business events that happen at specific points in the day—for example, the stock market has an opening and closing bell. For these scenarios, a **control service** can implement the schedule and broadcast time-based domain events, as shown in the following diagram. Downstream services can then react without duplicating the scheduling logic:

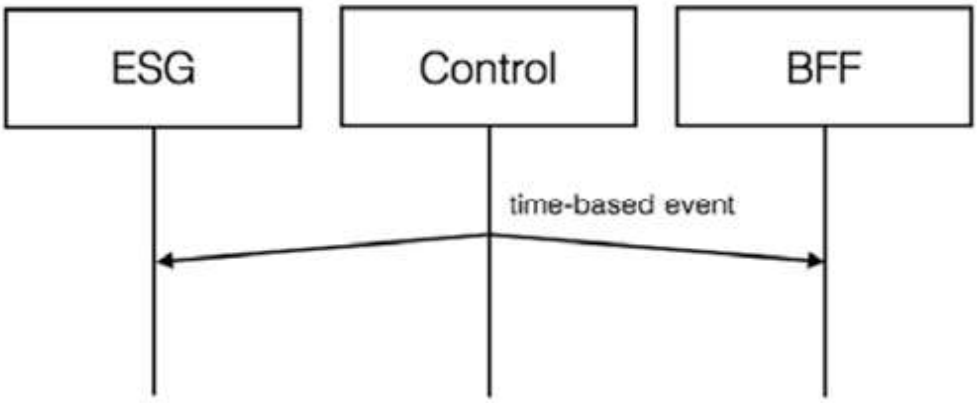


Figure 9.13: Cron sequence diagram

As with all autonomous services, we deploy these scheduled jobs to multiple regions to ensure that there is coverage in the case of a regional disruption. However, we do not want these scheduled jobs to be executed in multiple regions and thus perform duplicate work, so one region will serve as the **primary** and the others as **secondary**. Since these events are time-based, the **eastern-most** region is usually the *primary*.

It is important to note that we schedule these jobs at design time, not at runtime—in other words, we are not dynamically generating the schedule based on the activity of the system. We can use this to our advantage to solve

the duplicate execution problem, by staggering a job across regions and leveraging *replication* and *idempotence* to ensure that it executes in only one region.

Inserting job records and events

As is normally the case for control services, a scheduled control service is not responsible for performing the actual work. Its job is to signal when it is time to perform the work. So, we will use **job records** to control when we produce **job events**, as depicted in the following diagram:

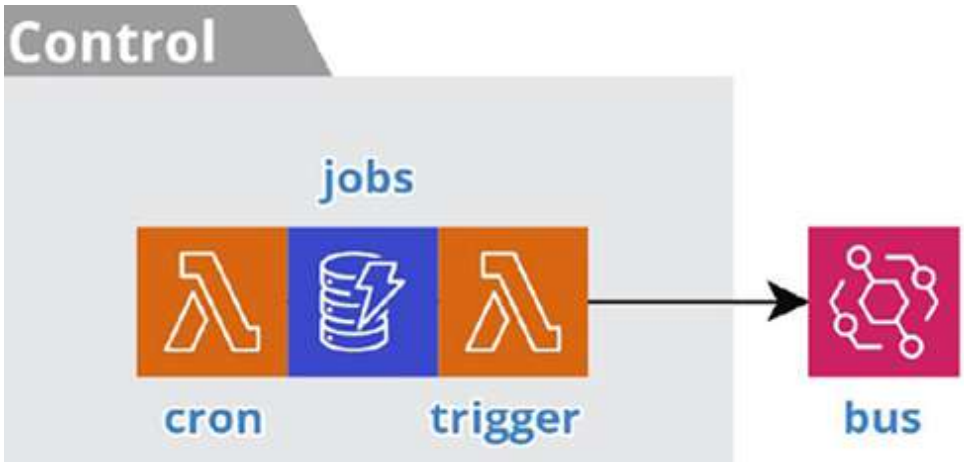


Figure 9.14: Regional cron job

We schedule the **cron** function to execute at the desired time, such as 8 a.m. in the eastern-most region. The function inserts a job record into the **jobs** table, such as AWS DynamoDB, with a **deterministic** identifier. It must be deterministic so that the cron functions in the other regions can attempt to insert the exact same record.

The eastern region is the first to insert the deterministic job record (more on this in a moment), so the **trigger** function is invoked with an **insert** event. The function asserts that the record was inserted from the current region and

publishes the appropriate time-based domain event to the **bus**. From here, downstream services take responsibility for performing the actual work.

Optionally, depending on the desired functionality, it may make sense to have the cron function call the regional health check service, which we discussed in the *Checking regional health* section. If the current region is unhealthy, then the function can decide not to insert the job record so that another region can initiate the job, which we cover next.

Relying on replication and idempotence

The **cron** function in each region will be invoked at the scheduled time and insert a deterministic job record, unless there is a regional disruption that prevents it. If the record already exists, then the *idempotence* of the deterministic insert will prevent the **trigger** function from being invoked, thus the time-based domain event will not be produced in that region and no duplicate work will be performed. But there is a catch—we have a **race** condition.

If we schedule all the cron functions for the same time, then they will all be the first to insert the job record into their region. To eliminate this **race** condition, we leverage the regional replication feature of a serverless database, such as AWS DynamoDB, and we **stagger** the jobs in the secondary regions to account for the **replication delay**. As a result, the job will typically execute in the eastern-most region until there is a disruption.

Keep in mind that this does mean that there will be a slight delay in the processing of the business functionality in a secondary region if there is a disruption in the primary region, but the aim here is to ensure that we

execute the functionality. In other words, *it is better late than never*. If you have more than two regions, then you will need to set up an order of precedence and stagger each successive region appropriately.

There is still a small chance that multiple regions will duplicate the work. For example, if there is a replication delay that is greater than the regional offset, then another region could be the first to insert the job record in its region. We mitigate this with our ever-present requirement to make all actions idempotent. So, make certain that the downstream logic uses the *domain event time*, not the *processing time*. This also highlights that the aim is to avoid the wasted effort of doing the same work in multiple regions, but not necessarily eliminate it. Trying to guarantee the elimination of this duplicate work is most likely a wasted effort in and of itself.

You can find a cron job example here: https://github.com/jgilbert01/templates/tree/master/template-control-service.

Running in multiple regions is a very important aspect of serverless systems, and the topics in this chapter are well worth the effort. Regional disruptions will happen, and it is very satisfying when your system fails over successfully with little to no interruption.

Summary

In this chapter, you learned why we should run our serverless systems in multiple regions and how we have been preparing for this all along. You learned how to implement a regional health check, configure regional routing, and set up regional data replication. You also learned how to dissect the regional failover flows, address the differences with external systems, and implement multi-regional cron jobs.

In the next chapter, we will see how to secure our autonomous subsystems in depth. We will secure our cloud accounts and CI/CD pipelines, along with all the layers of our subsystems from the perimeter and frontend down through our autonomous services.

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.

**Limited Offer*

10

Securing Autonomous Subsystems in Depth

In *Chapter 9*, *Running in Multiple Regions*, we covered how we can use cloud regions as another *bulkhead* to protect our systems from cloud provider disruptions. Running in multiple regions also improves performance for geographically dispersed users and provides a safety net that teams can leverage for regional canary deployments.

Now we turn our attention to securing our autonomous subsystems and services. Security itself is a kind of bulkhead against outside intruders. We will use cloud accounts as a bulkhead between autonomous subsystems and secure them and our CI/CD pipelines against attacks. We will learn how to secure the perimeter of the system along with the presentation and service tiers. We will also redact sensitive data and audit the system for compliance.

In this chapter, we're going to cover the following main topics:

- Shared responsibility model
- Securing cloud accounts
- Securing CI/CD pipelines
- Securing the perimeter
- Securing the frontend
- Securing BFF services
- Redacting sensitive data
- Securing ESG services
- Auditing continuously

Shared responsibility model

The security of our systems is of utmost importance. Yet there are countless stories of newsworthy data breaches. None of us want our systems to be in the headlines for this reason, but doing security right is hard. It is a complex, multidimensional, and multidisciplinary problem.

In our efforts to reduce lead times and increase our rate of innovation, market pressures may tempt us to cut corners that will come back to bite us. Fortunately, *serverless* and the *cloud security shared responsibility model* help us reduce lead times while also increasing security.

You may be thinking, how can we possibly do less, but achieve more? But we can achieve more by letting the cloud provider do the non-differentiating aspects of security. Then, we can re-focus our efforts on the security dimensions that far too often get the short end of the stick, such as redacting all the **Personal Identifiable Information** (**PII**) data.

The **shared responsibility model** essentially draws a line across the technology stack where everything below the line is the responsibility of the cloud provider and everything above the line is our responsibility. **Serverless** raises the bar by drawing that line higher in the stack so that we are pushing more security tasks below the line. The complexity remains, but now we get to apply our skills where we can achieve the most value.

There is still plenty for us to do. For example, we need to make use of the available cloud services and their security features. The following diagram depicts the scope of these services, which we will cover throughout this chapter.

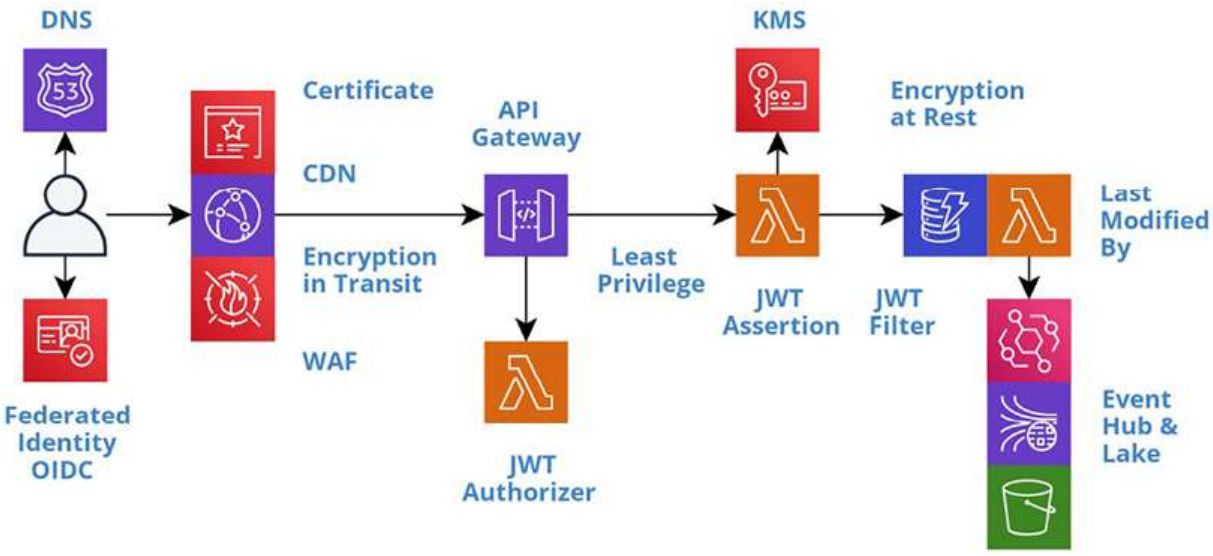


Figure 10.1: Security in depth

Let’s start by securing our cloud accounts first.

Securing cloud accounts

Securing our cloud accounts is not the first thing we usually think about when we think about security. We usually think about the application security topics we will cover in the later sections of this chapter, such as authentication, authorization, and encryption. But if you step back and think about it, all our autonomous services run in our cloud accounts. So, if we do not focus first on securing our cloud accounts, then we have essentially just secured the front door and have left the back door wide open for attack.

There was a story that I read in 2014 about a security breach at a company named *Code Spaces* that drove home to me the importance of securing our cloud accounts. If you have never heard of this company, it is because the breach put it out of business. A hacker took their cloud account for ransom and when the company fought back, the hacker deleted everything in the account, including the backups, which were housed in the same account.

It was a hard lesson to learn from, particularly for that company, but it taught us that we can easily solve this problem with a little due diligence. This was when I first started thinking about the idea of **autonomous subsystems** that each live in their own cloud accounts so that a breach in one subsystem would not incapacitate the entire system. I also went to work on replicating all my backups into an isolated account for recovery from a catastrophic disaster. Plus, I turned **multi-factor authentication** (**MFA**) on in all my cloud accounts the same day I read that article.

So, before we dive into the application security topics, let's cover some best practices for securing our cloud accounts, including treating accounts as code, systemwide policies, and identity access management.

Accounts-as-code

In *Chapter 2*, *Defining Boundaries and Letting Go*, we introduced the concept of autonomous subsystems and discussed ways of dividing a system into subsystems with well-defined boundaries. We also place each subsystem in a dedicated cloud account, because accounts create a natural bulkhead around the resources in the account. This allows autonomous teams to experiment without inadvertently affecting the resources of another team. This separation has the added security benefit of limiting the blast radius if a hacker gains access to an account.

However, we now have many cloud accounts that we need to create and manage. Plus, we need production and non-production accounts for each subsystem and shared accounts such as for backups and users. Furthermore, there are a variety of cross-cutting concerns that we need to configure in each account, such as auditing, security policies, observability, and so on.

We do not want to manage these by hand, because honest human error will likely result in the creation of security vulnerabilities.

Our cloud accounts are part of our infrastructure, so we will follow the same *infrastructure-as-code* practices that we use for our services. I refer to this as *accounts-as-code* to help highlight the importance of governing our cloud accounts. However, autonomy is crucial for creating a flexible architecture. We don't want teams to have to reinvent these wheels, but we also want to give teams the flexibility to innovate in these areas. As always, it is a balancing act between governance and flexibility. At the very least, every new subsystem needs a jumpstart.

We will follow a two-pronged approach that balances centralized control with decentralized flexibility. We will set system-wide policies and push mandatory configurations to all the accounts. Then we will let the autonomous teams pull recommended configurations into their accounts on an agreed-upon schedule. We will also provide a procedure for teams to recommend changes to the systemwide policies and mandatory configurations.

For example, a central team can own a master account and utilize AWS Organizations along with the open-source tool AWS Organization Formation (`https://github.com/org-formation/org-formation-cli`) to automate the creation and management of all other accounts. It would bootstrap the accounts with essential configurations, such as audit logging, automated backups, networking, and SSO. It would also seed common capabilities that all autonomous subsystems need, like an event hub and a regional health check service, and allow teams to tune these as needed.

Dedicated teams with the right knowledge and skills will handle the development of these cross-cutting concerns. However, the owners of each

autonomous subsystem need control over when to apply changes to their accounts. They must have the flexibility to override and/or enhance features as their circumstances dictate. A good approach is for each subsystem to fork its repository for each of the cross-cutting concerns. This will give each team its own CI/CD pipelines and the flexibility to pull in changes when they are ready and to submit pull requests back with suggested enhancements. And of course, all the same best practices that we will cover in *Chapter 11*, *Choreographing Deployment and Delivery*, apply to the governance of these stacks as well.

Let's pay particular attention to the cross-cutting concern of identity access management.

Identity access management

Securing our cloud accounts starts with identity access management. If you have a personal cloud account, then you are probably the super user of that account. If you work at a start-up, then your identity access management approach is probably still evolving.

If you work for a large enterprise, then you likely have an Identity Provider with **Single sign-on** (**SSO**) enabled and well-established roles and responsibilities. Regardless of your situation, here are some important practices you should understand and follow:

1. First, your **production** environment must be a **separate account** from your **non-production** environment. This creates a natural bulkhead between the environments and makes it easier to control access to the production resources while giving developers the freedom to experiment in the non-production account.

2. Second, we need a **well-defined set of roles** and permissions. These will typically include a mix of the following:
    - **Super User** – This role can do pretty much anything within an account. It is restricted to a small set of users who are usually members of a centralized security team that is responsible for implementing many of the cross-cutting security requirements.
    - **Power User** – This role usually has permissions for everything except for the likes of identity access management and network administration. In non-production accounts, we usually assign this role to the developers so that they are free to experiment. In production accounts, we assign this role to a limited set of operators, so that they can step in when there is a problem that our deployment scripts do not currently handle.
    - **Read-only** – This role has permission to read everything but change nothing. We assign this role to developers in the production accounts, so that they can help observe and operate the system.
3. Third, **multi-factor authentication** (**MFA**) is essential. Without MFA it is too easy for us to make the honest mistake of accidentally divulging a password or access key that gives hackers access to our system, like the story above. There are many MFA options that you can evaluate, but I recommend you **start right away** with one of the free smartphone apps, while you investigate enterprise alternatives.
4. Last, but not least, we need to practice good **access key hygiene**. Developers need access keys to non-production accounts so that they can test the deployment scripts they develop. These keys should not be shared, they should be rotated periodically, and they should require MFA as well. Here is an example of a utility that helps developers exchange their access key and an MFA token for a temporary key:

https://www.npmjs.com/package/aws-get-session-token.

Now, let's look at how we can secure our CI/CD pipelines.

Securing CI/CD pipelines

Our serverless systems are composed of cloud resources, such as functions, datastores, and messaging channels, and we use the cloud provider's deployment service, such as AWS CloudFormation, to automate the creation of these resources. In *Chapter 11*, *Choreographing Deployment and Delivery*, we will cover the *GitOps* flow we follow for deploying these resources via our CI/CD pipelines. But first, we need permission to create, update, and delete these kinds of resources. Plus, we typically use SaaS CI/CD services, such as GitHub Actions or GitLab-CI, and these services execute outside our cloud accounts.

So, we need an approach that limits the permissions that we give to these tools. We accomplish this by creating separate roles for the **pipeline service** and the **deployment service**. We will also introduce the concept of permission boundaries.

You can find a template for securing your CI/CD pipelines here: https://github.com/jgilbert01/templates/blob/master/template-pipeline-resources. This pipeline-resources stack owns resources used by the CI/CD pipelines.

Pipeline permissions

We generally think of our CI/CD pipelines as a single unit, but they are composed of multiple tools working in tandem. We will use this to our

advantage by giving these tools different permissions. The first tool in order of execution is the CI/CD service. The interesting thing is that this service doesn't actually perform the deployment; it only orchestrates the deployment. It supplies the glue between the Git service and the *Deployment service*. It reacts to the events emitted by the Git service and executes the deployment service.

This is a good thing because it means the CI/CD service doesn't need many permissions. It is the weakest link in the chain because it lives outside our cloud accounts. This means we will have to give it an access key, which it could compromise. So, we need to minimize the permissions that we grant it.

The pipeline only needs permission to upload the build artifacts and kick off the deployment service. We assign these permissions to a role named *DeploymentActionRole*. Then we create a *CiCdUser* with no permissions. Instead, we grant permission from the role to the user. This is a subtle difference, but it makes a compromised access key useless if the hacker doesn't know what role it can assume. Note, if your pipeline is executing within your environment, use the same approach by allowing the EC2 instance role (for example) to assume the *DeploymentActionRole* instead.

We need to generate an access key for the *CiCdUser* and store it as a secret in the CI/CD tool. Ensure that any necessary options are set to mask the access key in the pipeline logs. You should rotate the key periodically. When the pipeline executes, it will use the key to assume the *DeploymentActionRole* and start the deployment.

The following `gitlab-ci.yml` fragment is an example of using secrets to assume the *DeploymentActionRole* before kicking off the deployment:

```
  variables:
AWS_ACCESS_KEY_ID: $PRD_AWS_ACCESS_KEY_ID
```

```
  AWS_SECRET_ACCESS_KEY: $PRD_AWS_SECRET_ACCESS_KEY
  AWS_ROLE: $PRD_AWS_ROLE
  script:
  - eval \"$(assume-role) sls deploy -r us-east-1 -s prd\"
```

In this example, we assign the environment-specific secrets to the standard variable names and evaluate them with the `assume_role` command before invoking the Serverless Framework. Here is the library that supplies the command: https://www.npmjs.com/package/aws-assume-role-cicd.

The **Serverless Framework** (**SLS**) generates and uploads the deployment artifacts. One of these artifacts is the deployment template, which defines the resources that compose the autonomous service. Finally, SLS kicks off the cloud provider's deployment service with a reference to the template and then polls the service to monitor the progress of the deployment.

Deployment service permissions

Now we are getting to the real meat of the deployment process. At this point, the cloud provider's deployment service takes over, such as AWS CloudFormation. This service automates the creation of the resources defined in the deployment template. It keeps track of the resources in a unit that we call a **stack**. Each autonomous service lives in its own stack. The deployment service compares the new template against the current state of the stack to figure out what resources it needs to create, update, and delete.

This is where we need to give the deployment service the necessary permissions to carry out the deployment. Fortunately, our autonomous service patterns make these permissions very predictable. We assign these permissions to a role named the *DeploymentExecutionRole* and give the

deployment service permission to assume this role. The following `serverless.yml` fragment tells the deployment service which role to assume:

```
provider:
...
iam:
deploymentRole: arn:aws:iam::123456789012:role/mysubsys-cfn-exec
```

This last point is important and subtle again because we are only giving deployment permissions to a *trusted* entity, the cloud provider's deployment service, and no one else. The *Super* and *Power* roles we covered earlier have permission to fix a deployment in a pinch, but we only assign these roles to a small set of users in the production accounts. The autonomous teams that own and deploy the autonomous services do not have direct permission to deploy the needed resources. They must go through the CI/CD pipeline, which helps us control access to these permissions and restrict them to the least privilege.

Our *autonomous teams* are now able to perform deployments and our *autonomous service patterns* define some scope around the kinds of resources these services need to use. Now, from a security perspective, we need to define some concrete boundaries around what these services can do.

Permission boundaries

The goal of an autonomous team is to move quickly and continuously deliver business value to users with a fast feedback loop. To achieve this, we cannot depend on other teams. We must be *self-sufficient*.

However, when it comes to security, this is where the rubber meets the road. A centralized security team must place limits on the privileges of the

autonomous application teams. Otherwise, a team could escalate its privileges beyond what they need and put the entire system at risk.

Yet, the centralized security team does not wish to be a bottleneck. They understand that they will impede innovation if they force all teams to come to them to assign the permissions for all the resources used throughout our services and subsystems.

This is where **permission boundaries** come into play. For example, the functions within a service need permission to access the service's datastore. However, they do not need permission to create the datastore and they do not need to access datastores outside the service or subsystem.

So, the centralized team can define a permission boundary policy with these restrictions but leave the fine-grained details to the teams, as shown in the following `serverless.yml` fragment:

```
 Effect: Allow
Action:
- dynamodb:Query
- dynamodb:UpdateItem
Resource:
arn:aws:dynamodb:*:123456789012:table/mysubsys-*
```

In this example, the central team is restricting the actions a service can perform and limiting the set of resources to which they apply. They are limiting access to resources within a given subsystem based on a naming prefix and wildcard convention. Then the autonomous teams can refine the permissions in the individual services' teams as shown in the following `serverless.yml` fragment:

```
 Effect: Allow
Action:
```

```
  - dynamodb:Query
  - dynamodb:UpdateItem
Resource:
Fn::GetAtt: [ EntitiesTable, Arn ]
```

Here, the autonomous team is further restricting the permission of a given autonomous service to just the table within the service. As we saw in the BFF pattern in *Chapter 6*, *A Best Friend for the Frontend*, the naming convention for a datastore starts with the subsystem and the name of the service, so this service-specific permission is within the limits on the permissions boundary. The following `serverless.yml` fragment tells the service which boundary it must comply with:

```
provider:
...
iam:
role:
permissionsBoundary: arn:aws:iam::123456789012:policy/mysubsys-b
```

The Serverless Framework adds this boundary policy to all the roles within the autonomous service's deployment template. The central team enforces this within the *DeploymentExecutionRole* because this role does not have permission to create any roles that do not have the permissions boundary policy attached.

The result is that we have the best of both worlds, the central team maintains control by setting boundaries and the autonomous teams are free to innovate within those boundaries.

Now, let's look at securing the perimeter of the system.

Securing the perimeter

Security-in-depth is a layered approach where each additional layer provides more measures that an intruder must penetrate before gaining access to the valuable data within a system. In *Chapter 6*, *A Best Friend for the Frontend*, we covered the BFF pattern and in *Chapter 7*, *Bridging Intersystem Gaps*, we covered the ESG pattern. These services represent the boundary of the system because they support external actors. Before requests reach these services, we want them to pass through a hardened perimeter that weeds out illegitimate requests before they can harm the system. Fortunately, for our serverless systems, this is an area where the cloud provider's global resources provide an amazing level of scalability and protection. Let's look at how we can leverage global/edge infrastructure, use federated identity providers, and encrypt data in transit.

You can find a template for global resources used across a subsystem here: https://github.com/jgilbert01/templates/blob/master/template-global-resources.

Global/edge infrastructure

Cloud providers deploy their global services, such as DNS, CDN, and WAF, to edge locations all around the globe. This allows us to send a request to the closest edge location first and then route the request to the nearest region. With numerous edge locations, this is a great place to absorb traffic.

Domain Name System (**DNS**) services, such as Route53, provide Service Level Agreements (SLAs) of 100% and have the capacity to absorb and mitigate **Distributed Denial of Service** (**DDoS**) attacks. We use **Content Delivery Networks** (**CDN**), like AWS CloudFront, to deliver all the

resources for our micro frontend presentation tier from the edge. Cache settings ranging from a few minutes down to just a few seconds can have a very positive impact on the performance of our BFF services during an attack. For example, we can set the `cache-control` header on a 403 response to as little as a few minutes so the CDN can serve similar requests from the edge.

In addition, a **Web Application Firewall** (**WAF**), such as AWS WAF, can work with the CDN to protect the system from common exploits, such as **SQL injection** (**SQLi**), **cross-site scripting** (**XSS**), and unwanted bots. At a minimum, we should deploy rules to cover the **Open Worldwide Application Security Project** (**OWASP**) top ten web application security risks (https://owasp.org/www-project-top-ten). For internal applications, we can set up WAF rules to only allow requests from the corporate network and VPN.

Managing WAF rules and keeping them up to date with the latest vulnerabilities is a full-time job and requires specialized skills. A central team should manage these rules for all subsystems. For large organizations, this can be an internal team. Regardless, it is a good idea to buy managed rules from the cloud provider or third-party vendors.

Encryption in transit

All calls to a BFF and ESG service must encrypt the data in transit. At a minimum, this is necessary to secure the JWT bearer token. A bearer token gives access to any bearer of the token, so we must not expose it in transit. We must encrypt all other sensitive data as well, so it is best to encrypt all calls.

Serverless services, such as AWS API Gateway, Lambda, and DynamoDB, require the use of HTTPS. Therefore, we only need to take steps to encrypt the calls to the CDN. This requires a custom SSL certificate. A cloud provider's certificate management service, such as **AWS Certificate Manager** (**ACM**), can create the certificate and automatically rotate it each year. Then we associate the certificate with the CDN and configure the CDN to require HTTPS.

Federated identity

We need to provide authentication and authorization support for our presentation tier and our boundary services, such as a SaaS federated identity provider like Okta or AWS Cognito. The identity provider will implement the **OpenID Connect** (**OIDC**) protocol and the output of a successful authentication is a **JSON Web Token** (**JWT**). The provider should also have **SAML 2.0** support for the corporate directory systems. We will cover the details in the following sections.

Each autonomous subsystem should maintain its own user pool and security configuration, even if they use the same SaaS identity provider. For example, our food delivery system has separate subsystems for customers, drivers, and restaurants. Each of these is a different pool of users, even though drivers and restaurant employees can also be customers. This gives each subsystem complete flexibility. For example, we probably want to allow social sign-in options for customers, but we may not want this for restaurant employees and drivers.

Now, let's look at securing the presentation tier.

Securing the frontend

In *Chapter 3, Taming the Presentation Tier*, we covered the concepts of micro frontend architecture. In this section, we will look at the security of the presentation tier. First, we will provide authentication with **OpenID Connect** (**OIDC**).

Then we will add role-based authorization with conditional rendering and routing. Finally, we will pass the **JSON Web Token** (**JWT**) along on BFF calls to help secure the service side.

OpenID Connect

User authentication is the first step in securing the presentation tier. We need to identify the user so that we can determine what we will allow them to see and do. Traditionally, teams spent many cycles building an authentication system. Today, we can accelerate delivery by leveraging the OIDC specification and SaaS identity providers.

OIDC is a simple identity layer on top of the **OAuth 2.0** specification. It defines the authentication flows and adds the identity token (that is, `idToken`). The identity token provides information about the user in the form of claims, such as the user's assigned groups or roles. `idToken` is represented as a JWT. We will use this information to secure the frontend and the backend.

The following code block shows an example of integrating a SaaS security SDK into the presentation tier:

```
import {useState, useEffect} from 'react'
import Auth from '@aws-amplify/auth'
Auth.configure({
```

```
    userPoolId: '...',
    redirectSignIn: window.location.origin,
    responseType: 'code'
    ...
  });
  export const useAuth = () => {
    const [state, setState] = useState({ isSignedIn: false,
      ... });
    useEffect(() => {
      Auth.currentAuthenticatedUser()
        .then((user) => setState({
          user, isSignedIn: true,
          groups: user.signInUserSession
            .idToken.payload['cognito:groups'],
          idToken: user.signInUserSession.idToken.jwtToken,
        }));
    }, []);
    const signIn = () => Auth.federatedSignIn();
    const signOut = () => Auth.signOut();
    const hasRole = (role) => state.groups?.includes(role);
    return { ...state, signIn, signOut, hasRole };
  };
```

This specific example creates a custom React Hook called `useAuth`. The Hook hides the interface of the specific SaaS security provider from the rest of the system. In this case, we are integrating AWS Cognito using the AWS Amplify SDK. However, the rest of the system will interact with the interface returned by the Hook. This will allow the team to change the security provider more easily.

The Hook lives in the `@my-subsys/auth` micro application that contains the various security utilities used across all the micro applications. It configures the SaaS SDK with the subsystem's information, such as the identifier for the specific user pool, where to redirect the user after the hosted login page completes, and which authentication flow to follow.

The Hook returns state information, such as `isSignedIn`, `idToken`, and `user`, along with wrapper functions, such as `signIn`, `signOut`, and `hasRole`. The custom Hook also creates a `useEffect` Hook that acts as a callback to handle the redirect from the hosted login screen and receive the user information. The following code block shows how to use the custom Hook to render sign-in and sign-out links:

```
import { useAuth } from '@my-subsys/auth';
export const SignInOut = () => {
  const { isSignedIn, signIn, signOut, user } = useAuth();
  return isSignedIn ?
    <SignOut user={user} signOut={signOut} /> :
    <SignIn signIn={signIn} />;
};
```

This component imports the `useAuth` Hook and calls it to retrieve the user state and the helper functions. It conditionally renders the `SignIn` and `SignOut` components based on the `isSignedIn` flag. It passes the `signIn` and `signOut` functions for the use of clicking on links. Once the user has signed in, the SDK handles the details of storing the user information and refreshing expired tokens.

Now that we have the user's information, we can use it to control access.

Conditional rendering

The user experience should follow the principle of least privilege. For a smooth user experience, the presentation tier should not present the user with menus and buttons that require higher authorization. We should conditionally render components based on the user's permissions. For

example, we should not render save and delete buttons if the user only has read access.

We have already seen conditional rendering based on the `isSignedIn` flag. The following code block shows role-based rendering:

```
import { useAuth } from '@my-subsys/auth';
export const SomeProtectedComponent = () => {
  const { hasRole } = useAuth();
  return hasRole('Admin') ? <div> ... <div/> : null;
};
```

This component uses the `useAuth` Hook to retrieve the `hasRole` function. Then it conditionally renders the content if the user has the `Admin` role. In this case, it renders nothing (that is, `null`) if the user does not have permission. In other cases, it may make sense to render alternate content.

We can also integrate role-based access into our mount points. The following code block revisits the mount points example from the *Dissecting micro frontends* section in *Chapter 3*, *Taming the Presentation Tier*:

```
import {useAuth} from '@my-subsys/auth';
import {useMountPoint} from '@my-subsys/main';
export const LeftNavMenuItems = () => {
  const {hasRole} = useAuth();
  const {items} = useMountPoint('left-nav');
  return items
    .filter((item) => hasRole(item.role))
    .map((item) => <Menu.Item ... />);
};
```

In this example, we add a filter on the metadata for the mount point. The metadata defines the required `role`, and we pass it to the `hasRole` function.

For each micro application, we include the required role in its manifest file as follows:

```
  "mount-points": {
     "left-nav": [{..., "role": "Admin", ...}, ...]
   }
```

In addition to conditionally rendering components, we must also control access to routes.

Protected routes

The user can manually enter a route into the browser's navigation bar to navigate to a page without clicking on a menu or button. For example, the user can bookmark a specific page. This will allow an unauthorized user to gain access to a page unless we secure the routes within each micro-app.

As we covered in the *Dissecting micro frontends* section of *Chapter 3*, *Taming the Presentation Tier*, the micro frontends framework handles top-level routing and then delegates routing to the micro application. From here, the routing rules within a micro-app can use the `isSignedIn` flag and the `hasRole` function to control access to content. The following code block shows how to create a `ProtectedRoute` component that we can include in our auth micro app:

```
import React from 'react';
import {Route, Redirect} from 'react-router-dom';
import {useAuth} from './auth';
const ProtectedRoute = ({component: Component, role,
    ...rest}) => {
  const {hasRole} = useAuth();
  return <Route {...rest} render={ props => hasRole(role) ?
        <Component {...rest} {...props} /> :
```

```
          <Redirect to='/' />
      } />;
};
export default ProtectedRoute;
```

This specific example uses `react-router`. The `ProtectedRoute` component wraps the unprotected `Route` component to override its `render` property. The new `render` function asserts that the user has the required role. If the user has permission, then we render the requested page component; otherwise, we render the `Redirect` component to send the user to the home page.

Now that we have the conditional rendering and routing, we need to pass the user's information along when we invoke a BFF service.

Passing the JWT to BFF services

In *Chapter 6*, *A Best Friend for the Frontend*, we covered the various responsibilities of BFF services and the important role they play in supporting micro applications. Many BFF services are only accessible to authorized users. To invoke a secured BFF service, the micro application must pass a JWT on each call to the service. The following code block shows an example of calling a secured BFF service:

```
import {useAuth} from '@my-subsys/auth';
const {idToken} = useAuth();
const {data, loading, error} = useFetch('/my/bff', {
  headers: {Authorization: 'Bearer ${idToken}'}
});
```

First, the micro application retrieves the `idToken` from the auth micro application that manages the information about the user's authenticated

session. Then the application includes the token in the request using the `Authorization` header field.

It is important to note that we should not pass an ID token to third-party services. They will require an access token instead. We will cover interactions with third-party services in the *Securing ESG services* section.

A BFF service, on the other hand, specifically supports a given micro application and provides the service-side functionality of the application. Therefore, we need to pass the `idToken` to a BFF, so that it can leverage the claims to perform assertions and filters, based on the user's entitlements.

Now let's turn our attention to securing the BFF services.

Securing BFF services

In *Chapter 6, A Best Friend for the Frontend*, we covered the BFF pattern. These services expose a synchronous interface to the frontend, which means they expose an attack surface at the boundary of the system. Fortunately, securing a serverless BFF service in depth is mostly a *declarative* exercise. We need to grant access to a service, then grant access to the data, track who did what, and ensure a service cannot access resources outside the service. Let's discuss these topics in this order.

JWT authorizer

When we left off in the previous section, *Securing the frontend*, we passed the user's `idToken` in a request to a BFF service using the `Authorization` header. Once a request reaches the BFF service, the first step is to authorize the request.

The API gateway performs this step, so that we separate the technical concerns of handling the JWT token from the business logic. This also eliminates unnecessary calls to the *query* and *command* functions when the request is *unauthorized*.

The authorization process verifies the *signature* of the token and checks the *expiration* time, at a minimum. The process will usually validate the *issuer* and the intended *audience* as well.

Verifying the signature requires access to the key used to generate the token. A third-party provider will expose its public key, following the **JSON Web Key Set** (**JWKS**) standard, with a URL like `https://my-identity-provider/.well-known/jwks.json`. We configure the API gateway with this URL, so it can perform the verification.

Alternatively, we can implement a custom authorizer function that verifies the token. Authorizer functions provide the flexibility to verify custom and non-standard tokens, as well as standard tokens. Choosing between the alternatives depends on the needs of the token and the features of the API gateway service.

The following `serverless.yml` fragment, from the BFF service template, configures the `rest` function to use the `authorizer` function provided by AWS Cognito:

```
functions:
rest:
handler: src/rest/index.handle
events:
- http:
...
authorizer:
arn: arn:aws:cognito-idp:REGION:ACCOUNT:
userpool/ID
```

If the JWT token is invalid, then the API gateway will reject the request with a 403 Forbidden status code. Otherwise, the request will proceed, and the API gateway will include the claims from the JWT in the request context when it forwards the request to the *query* and *command* functions. From here, we can perform JWT assertions and JWT filters as needed.

JWT assertion

So far, we have used JWT to authorize access to a service as a whole. This may be all that is necessary for some services. For others, we may need to use the claims in the token to grant role-based authorization to the individual queries and commands in the service as follows:

1. First, we need access to the claim that contains the user's assigned roles. The following code shows how to access the `groups` claim that AWS Cognito includes in its JWT token:

```
const getUserGroups = (request) => request
  .requestContext.authorizer.claims['cognito:groups'];
```

2. Next, we need the code to assert that the user has the required role. The following middleware code asserts that the user is a member of a group with the required role name before continuing. Otherwise, it returns a 401 HTTP status code in the response:

```
export const forRole = (role) => (req, res, next) => {
  const groups = getUserGroups(req);
  if (groups.includes(role)) return next();
  else res.error(401, 'Unauthorized');
}
```

3. Then we add the middleware to the individual routes and specify the required role name, such as `forRole('Admin'):`

```
api.get('/things', forRole('Admin'), getThings);
```

We can implement the same role-based assertions in GraphQL by decorating the model with directives, such as `@hasRole`:

```
extend type Query {
  things(...): [Thing] @hasRole(role: 'Admin')
 }
```

We can also implement fine-grained assertions in the `model` class if need be. However, we should pass the claims to the model in a normalized format to decouple the model from the chosen authorizer.

JWT filter

So far, we have used JWT to authorize access to a service and used the claims in the token to assert role-based authorization to the queries and commands in the service. Now, we need to control access to the data at the **record level**. For example, in our food delivery system, customers should only have access to their own orders, and restaurant employees should only have access to their restaurant's menu.

To accomplish record-level security, we need to carefully craft the **queries** that grant users access to the data. A request may explicitly include the data for these filters in the URL query parameters. However, **we cannot trust these values**, because they can be set to any arbitrary value by the caller.

Instead, **we need to use the claims from the JWT to set the filter values**. For example, the token will include the subject's identity and should include the groups the subject is a member of. We can trust these values because **we have asserted the authenticity of the token signature**. In other words, the caller cannot tamper with the contents of the signed token. You should also assert the claims against the requested values to help find bugs or suspicious activity.

Last modified by

Auditing is another common security requirement. We need to track **who** changed **what** data and **when**. We will dive deeper into auditing shortly, in the *Auditing continuously* section, but first, we need to **record** who invoked a **command** on a BFF service.

This information is available in the JWT. The following code block provides an example of accessing the username from a JWT token in an AWS Lambda function:

```
const getUsername = event => event
  .requestContext.authorizer.claims.sub;
```

In the `rest` or `graphql` handler function, we use `getUsername` to pull the username (that is, subject ID) from the claims and pass it to the constructor of the `model` class in a normalized format:

```
export default class Model {
  constructor(connector, username) {
    this.username = username;
    ...
  }
  save(id, input) {
```

```
      return this.connector.save(id, {
        ...input,
        lastModifiedBy: this.username,
        });
    };
  }
```

The methods in the model class assign the `username` to the `lastModifiedBy` field. Then the *change events* will record who performed the *command* when the `trigger` function emits these events for the inserts, updates, and deletes.

Least privilege

As always, we should assign permissions to the least privileges necessary. In addition to assigning permissions to users, we must also assign permissions between resources. For example, we must grant the API gateway permission to access the functions in a service and we must grant the functions permission to access the datastore and the event hub.

The Serverless Framework will automatically generate any permissions it can implicitly infer from the specified metadata. For example, it can infer that the API gateway will need permission to access the `rest` function, whereas we have to explicitly grant permissions for the resources accessed within the function code.

The following `serverless.yml` fragment, from the BFF service template, grants the `rest` function permission to query and update the datastore:

```
 iamRoleStatements:
- Effect: Allow
Action:
- dynamodb:Query
- dynamodb:UpdateItem
```

```
  Resource:
  Fn::GetAtt: [ EntitiesTable, Arn ]
```

Here we are granting access to the Entities Table. This table is owned by the service and lives in the same stack. This means that the ARN for the table is readily available and there is no need to use wildcards. Do not use wildcards when granting permissions, because it increases the potential attack surface.

Now let's turn to the ever-important topic of securing data at rest.

Redacting sensitive data

Defense in depth means that every layer of a system must do its part to prevent an attack. **Encrypting data at rest** is the last line of defense, and yet it is where the most shortcuts are taken, as evidenced by all the high-profile data breaches. If a hacker breaches all the other layers of security, the data will still be secure if we have properly **redacted** the sensitive information at the application level.

To truly secure data at rest, we must do more than just turn on the datastore's encryption feature.Disk-level encryption only secures the data when the disk is removed from the system. While a disk is attached to the system, the data is automatically decrypted when accessed with sufficient privilege. Unfortunately, this is what happens in most data breaches. A hacker gains privileged access through an alternate channel and the data is automatically decrypted when it is read from the datastore.

To prevent this, we must redact (that is, encrypt) sensitive data at the **application level**. This will ensure that sensitive data is unusable in the inevitable event of honest human error that inadvertently opens a hole that results in a data breach. But, as the domain experts of our services, teams

must take responsibility for identifying and redacting sensitive information. Let's look at how teams can use *envelope encryption* to secure their domain models by redacting sensitive data.

Envelope encryption

Envelope encryption is the practice of using multiple layers of keys and encrypting one key with another key. We encrypt sensitive data with a **data key** and then we encrypt the data key with a **master key**. Then we store the encrypted data and the encrypted data key together to associate the data key with the data.

When we need to decrypt the data, first, we decrypt the data key with the master key. Then, and only then, can we decrypt the data with the decrypted data key. Without privileged access to the master key, the sensitive data remains redacted.

Let's take an example where the Person domain entity contains `PII` in the name field. Therefore, we must encrypt this field in all events and whenever we save the field in a datastore. The following data block shows a domain event with an encrypted `name` field and an `eem` field that contains the envelope encryption metadata that we need to decrypt the object. The `eem` field includes the list of encrypted fields, `masterKeyAlias`, and the encrypted `dataKeys`:

```
{
  ...
  type: 'person-updated',
  person: {
    ...
    name: 'zzzzzzzzzzzz=',
  },
  eem: {
```

```
      fields: ['name'],
      masterKeyAlias: 'alias/my-master-key',
      dataKeys: {
        'us-east-1': 'xxxxxxxxxxxx=',
        'us-west-2': 'yyyyyyyyyyyy=',
      },
    },
  }
```

Note that we encrypted the data key twice, once per region. We need to do this when the cloud provider's key management service is regional instead of global. This means that we must declare a master key in each region and encrypt the data key with each master to ensure that we can still decrypt if a region is unavailable. The **aws-lambda-stream** library relies on the **aws-kms-ee** library (https://www.npmjs.com/package/aws-kms-ee) to handle all these details.

The following code extends the example we used in *Chapter 5*, *Turning the Cloud into the Database*, for populating a materialized view in a listener function:

```
import { materialize } from 'aws-lambda-stream';
const RULES = [{
  id: 'm1',
  pattern: materialize,
  eventType: /person-(created|updated)/,
  eem: { // envelop encryption metadata
fields: ['name'],
    masterKeyAlias: process.env.KMS_MASTER_KEY_ALIAS,
    regions: process.env.KMS_REGIONS,
  },
  toUpdateRequest: async (uow, rule) => ({
    Key: ...,
    ...updateExpression(await rule.encrypt({
      ...uow.event.person,
      ...,
    })),
```

```
    }),
  }];
```

First, the `listener` function uses the `eem` data contained in the event above to decrypt the data before we pass it to the pipelines. Then we must re-encrypt the data before we save it to the datastore. To do this, a rule declares an `eem` field with the metadata needed to encrypt the object, including the list of fields to encrypt, `masterKeyAlias`, and the list of regions for which we will encrypt the new data key.

The rule also exposes an `encrypt` function that we use to perform the encryption. The encryption process returns an **eem** structure, just like the one contained in the `person-updated` domain event example, that contains the new encrypted data key. We save this metadata in an `eem` field alongside the domain object in the same database record so that we can decrypt the object later.

This process works in a similar manner when we process sensitive data in a `trigger` function. The `trigger` function decrypts the data it receives from the CDC stream using the data key stored in the `eem` field. Then the pipeline uses the metadata provided in the rule's `eem` field to re-encrypt the data in the domain event before publishing. Likewise, a `command` function must encrypt sensitive data before saving and a `query` function must decrypt the data before returning it to the client.

It is important to rotate the various keys. A key management service, such as AWS KMS, will rotate the master key on a yearly basis. These services retain all the previous versions of a master key so that we can decrypt existing data keys, but we encrypt all new data keys with the latest version of the master key. We should generate a new data key each time we save a domain object with sensitive data to a datastore or publish it in a domain

event. This restricts the blast radius of a compromised data key to a single domain object.

Crypto-shredding

The **General Data Protection Regulation** (**GDPR**) is an EU law on data protection and privacy. One article of the law states that a subject can request the erasure of their personal data. At first glance, this requirement would seem to be difficult to achieve in an event-first system that leverages the systemwide Event Sourcing and CQRS patterns and maintains an event lake per autonomous subsystem. However, data life cycle architecture and envelope encryption provide a simplified solution.

First, as data moves through its life cycle, it should remain as **lean** as possible. If a service does not need PII data, then it should not hold a copy. If a service does need to handle PII data, then it should hold as little as possible, keep the TTL as short as possible, and use envelope encryption. And analytics services should de-identify all data.

Next, we need a solution for long-term data, such as archived data and the events in the event lakes. We want this data to continue to be useful, but we need a way to essentially **de-identify** the PII elements it contains. **Crypto-shredding** provides a solution that renders data useless without deleting the data. We simply need to delete the data key that we used to perform the envelope encryption.

To support this approach, we add another layer of keys to the envelope encryption process. We still have a master key, and we still create a new data key every time we encrypt data, but we encrypt the data key with a **subject-specific key**. We can think of this additional key as a subject-specific master key. When a subject asks for their data to be erased, we just need to delete

the subject-specific key. This effectively de-identifies all the subject's data because all the sensitive data can no longer be decrypted.

Finally, we rely on the inversion of responsibility principle to help control the scope and complexity of the erasure process. When a subject requests erasure, we produce a stream-first event to record the fact the subject made the request. From here, downstream services take responsibility for performing the necessary steps, such as deleting the subject-specific key that they own.

Now let's turn our attention to securing ESG services.

Securing ESG services

In *Chapter 7*, *Bridging Intersystem Gaps*, we covered the ESG pattern, which provides an anti-corruption layer around external systems. The *syntactic* details of various security technologies often differ between systems. ESG services encapsulate these details so that they do not pollute the rest of the system. We may even isolate legacy systems and open APIs in their own cloud accounts to help control the attack surface that exists in the interactions between disparate systems. In this section, we will address shared secrets such as passwords and access keys, as well as API keys.

Securing shared secrets

The various ESG scenarios require connecting to external resources. We secure these interactions in transit with SSL, but first, they must be authenticated. Legacy systems will most likely require a *username* and *password*, while modern systems typically require some sort of long-lived *token*. For example, a SaaS system may use API keys and a cloud provider

may require an access token. These are all semantically the same; they are all *secrets*. The syntactical details only really matter to the external system. What matters to an ESG is that the necessary secret is readily available and that it stays a secret.

It is important to note that we are not authenticating users in these scenarios; we are authenticating systems. In the BFF pattern, an end user is authenticated at the beginning of a session and the token they receive is short-lived. In the ESG pattern, the token is **long-lived** and periodically **rotated**. For a legacy system that uses username and password, we can think of these as system users.

The ESG pattern encapsulates the details of interacting with secrets. A secret must be encrypted at rest and the ESG service should have the least privileged access to the secret. To accomplish these requirements, a secret should be stored using a fully managed cloud service, such as AWS Secrets Manager. The ESG function will retrieve the secret, cache it in memory, and use it when connecting to the external system. The actual value of the secret must never be logged. The following code extends a `listener` function to initialize secrets:

```
import {defaultOptions, getSecrets,
...} from 'aws-lambda-stream';
...
export const handle = async (event, context) => {
  const options = await getSecrets(defaultOptions);
  return new Handler(options).handle(event)...;
};
```

In this example, we use the `getSecrets` utility function to retrieve and cache the secrets, and then we pass them to the handler as `options`. From the standpoint of the various pipelines, the secrets just look like any other

option. How you put the secrets into the secrets manager will depend on your environment. One approach uses the following plugin to copy the secrets during deployments: https://www.npmjs.com/package/serverless-secrets-mgr-plugin.

Secrets are long-lived, but they should not be perpetual. It is best to rotate them periodically. A cloud service, such as AWS Secrets Manager, will provide a rotation feature with scheduling and the ability to implement custom code that calls the external system to create a new secret. A typical rotation algorithm might add a new secret every 90 days, disable the previous secret several days later, and delete it several days after that. For username/password secrets, we can alternate between two username/password pairs. This allows in-flight transactions to continue to work and supports rolling back to the previous secret if needed.

Using API keys

In open API and third-party webhook scenarios, an ESG service exposes endpoints on the public internet. In the BFF pattern, we secure these endpoints with an *OIDC JWT* that represents an end user's authenticated session. For these ESG scenarios, we use an API key to identify the specific external system making the call, not the end user.

An API key grants an external system access to a specific interface. Therefore, it is best to limit the features exposed by each interface to ensure that an external system has the least privileges necessary. The developer portal of a SaaS product should provide the ability to request and revoke a key for each API. An API gateway service should provide the capability to define usage policies that protect interfaces by throttling access with rate

limits and upper limits per API key. Tiered usage policies may also be applicable, such that freemium accounts have lower thresholds and so forth.

It is important to keep in mind that API keys are long-lived bearer tokens. If they are stolen, they can be used to access resources until they are revoked. Therefore, we must secure these secrets as we discussed previously. We should also rotate these keys, so your open API and developer's portal should provide this capability.

Now let's turn our attention to continuous auditing.

Auditing continuously

In *Chapter 2*, *Defining Boundaries and Letting Go*, we covered the topic of governance and the impact it can have on the speed of innovation. Governance covers a wide spectrum of subjects, but by and large, we focus on security and the compliance of a system with security regulations, such as NIST, PCI, GDPR, and HIPAA, plus internal policies, and best practices. Throughout this chapter, we have covered how to secure our serverless systems in depth and by design. Now we need to ensure that we are doing so and not straying off course.

Continuous auditing is an approach to governance that allows us to put guardrails in place without impeding the speed of innovation. We will look at automating audits at *build-time* and *runtime* and then look at how our event-driven architecture helps us audit the users of the system as well.

Build-time and runtime auditing

Confidence and automation are prerequisites for continuous deployment and delivery. Teams that are unsure about the security of their system will slow

down and manual security audits are just plain slow. **Continuous auditing** solves these problems by automating security audits and integrating them into our methodology and product delivery pipeline so that we can **find and fix issues fast in real time**.

We start by creating **templates** for each of the major patterns so that teams can *seed* new features that are compliant from the start. For example, the first pull requests for a new repository should seed a complete and compliant CI/CD pipeline that deploys a skeleton all the way to production to verify that all the governance plumbing is in place.

Modern build pipelines have become very robust, thanks to strong open-source community support. Dependency management tools, like NPM, have built-in security auditing to identify dependencies with known vulnerabilities. Open-source static code analysis rules do a great job of enforcing well-established design patterns and security best practices. Plus, we can plug in commercial scanning tools at the same point in the pipeline to ensure that we are scanning for the latest known threats. At any point, we can **fail the build** and fix the problem before deploying the code.

We can check the system for vulnerabilities at **runtime** as well. For example, *AWS Config* can assert that you have configured your cloud resources properly and remain so. *AWS Inspector* can scan your code for vulnerabilities after deployment and after a new vulnerability has been identified. *AWS Macie* can scan your datastores to find sensitive data that you have not properly encrypted. It is also a best practice to automatically run *penetration tests* after each deployment instead of the traditional quarterly schedule.

Of course, we must pair auditing with **mentoring**. Teams need to understand the rules and best practices before they can comply with them. In *Chapter*

12, Optimizing Observability, we will add monitoring to the entire product delivery pipeline, to provide us with timely, actionable information so that we know when to jump into action. We can also leverage observability metrics to help identify teams with additional mentoring needs. Continuous auditing is a cross-cutting concern, so a centralized team with all the specialized skills can define the auditing policies for all the teams.

We need to audit more than just the software of our system. Now, let's look at how we can audit the use of the system.

Change event audit

Security requirements typically include the need to audit **who** did **what**, and **when**, within the system. Not so coincidentally, our event-driven architecture is already doing this for us. In *Chapter 4, Trusting Facts and Eventual Consistency*, we covered the *Systemwide Event Sourcing* pattern and how events and the *Event Lake pattern* provide a complete audit trail of the activity within a system. Let's do a quick review of what we are already doing and see how we can put this information to good use.

First, when a user performs a `command` in the system, we store the user's ID in the `lastModifiedBy` field of the affected rows, as we covered above in the *Securing BFF services* section. Next, we have the change data capture feature of the datastore to notify the system whenever the data is changed, as we covered in Chapter 5, *Turning the Cloud into the Database*. Finally, the trigger function in the BFF service produces events to the event hub, as we covered in *Chapter 6, A Best Friend for the Frontend*. In addition to any higher-level domain events, a BFF service should also produce the lower-level change events with all the *raw* details of the *before* and *after* images of the data.

With all these change events available in the system, we can use our BFF and Micro App patterns to give users visibility into the activity of the system. We can create an **audit service** that consumes these events and materializes the data in a format that allows us to see who is doing what in the system. This might include the *id*, *discriminator, timestamp*, and *lastModifiedBy* fields and a list of the fields that the user changed along with their before and after values. We might allow a user to access the audit by domain entity ID and/or we might supply a report for a given data range, like we covered in the *Implementing different kinds of BFF services* section of *Chapter 6, A Best Friend for the Frontend*.

All in all, auditing is a great example of the beauty of event-driven architecture and the *Inversion of Responsibility* principle that we covered in *Chapter 2, Defining Boundaries and Letting Go*. Upstream services just produce events when their state changes, without concern for how downstream services may use the events. This opens a world of opportunity for innovation downstream as we discover new uses for the events and data.

Summary

In this chapter, you learned about the shared responsibility model for cloud security and why it is important to secure our cloud accounts and CI/CD pipelines first. You learned how to secure the perimeter of the subsystem with edge services along with securing the presentation tier and the backend services with OIDC and JWTs. You also learned how to use envelope encryption to redact sensitive information and how to secure secrets. Finally, you learned how to audit for compliance.

In the next chapter, we will see how continuous deployment and continuous delivery work together to facilitate continuous innovation. We will cover

multiple levels of planning, CI/CD pipelines, continuous testing, zero downtime deployment, and the robustness principle.

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.

**Limited Offer*

11

Choreographing Deployment and Delivery

In *Chapter 10, Securing Autonomous Subsystems in Depth*, we covered securing our cloud accounts and CI/CD pipelines. We secured the perimeter, the frontend, and our BFF and ESG services. In addition, we redacted sensitive data and saw how the shared responsibility model enables teams to focus their effort on business logic.

We will now turn our attention to the methodology of continuously building, testing, deploying, and delivering functionality to end users. In this chapter, we will focus on the best practices for keeping things from going wrong in the first place. You will learn how to choreograph the continuous deployment and delivery of services and applications with zero downtime and with the confidence that all the necessary checks and balances are in place.

In this chapter, we're going to cover the following main topics:

- Optimizing testing for continuous deployment
- Focusing on risk mitigation
- Achieving zero-downtime deployments
- Planning at multiple levels
- Turning the crank
- Dissecting CI/CD pipelines

Optimizing testing for continuous deployment

In *Chapter 1, Architecting for Innovation,* we stated that the purpose of software architecture is to enable change. We want to continuously deliver innovations to our end users and produce business value. To meet this goal, we strive to reduce lead times so that we can increase the rate of feedback from end users and create knowledge about what works and what does not. To this end, we have focused throughout this book on optimizing our architecture for short lead times.

Now, we need to turn our attention toward optimizing our testing processes. Traditional testing assumes that we know what is going to work, and we can gather all the requirements and acceptance criteria upfront. Then, we assert that the software is correct before we deploy and release it to the end user.

However, our traditional testing practices work against our stated goal. They are optimized for large batch sizes, which increases deployment risk. They encourage longer lead times, which delays feedback.

So, testing is the last piece of the puzzle that we need to change to support short lead times. We need to turn our testing practices on their heads. We need to turn testing into a process of continuous discovery.

Continuous discovery

Another way of stating our goal is that we are aiming for **continuous discovery**. We need to formulate hypotheses, perform experiments, and iteratively discover the best solutions for our end users. In other words, we need a testing process that helps us build and discover the right solution, not one that asserts that we are building to a preordained specification.

To gather feedback and discover what works, we need to put the software in the hands of the end user in the production environment. This means that we need to deploy much more frequently. Our testing processes must help us to do so safely.

We need to push changes to production that are incomplete, but we need to be certain that there are no regressions. We need to assert that the system is healthy and learn which features users are actually using. To accomplish this, we certainly don't want less testing. We need more testing—a lot more testing.

As an example, let's look at the statistics for a prototypical project. We delivered 250 stories over 18 months. In the end, the code base included over 6,000 automated test cases, which accounted for well over 50% of the code.

That certainly seems like a lot of testing, but *we built up the test suite incrementally*. We completed the stories with approximately 3,000 tasks, and each task resulted in a production deployment. On average, we completed eight tasks (that is, deployments) per day and wrote two test cases per task.

In other words, testing is not a separate task—it is an integral part of each task. Testing is the safety net that gives us the confidence to forge ahead, continuously deploy, and discover the right solution. Testing is continuous.

Continuous testing

We are changing our testing mindset to one that allows us to focus on iterating to the right solution. We are no longer waiting until the end to perform testing—we are testing all the time. After all, we are experimenting with functionality, which is a form of testing. Experiments help us eliminate

the wasted effort of building the wrong solution, but we also need testing that gives us the confidence to move forward with our experiments.

Continuous testing is the overarching term for the many types of testing we perform. The following word cloud highlights the scope and breadth of the continuous testing effort. It presents the relative relationships between the testing techniques and approaches that we bring to bear to help give us the confidence to perform more frequent deployments:

Figure 11.1: Continuous testing: word cloud

Figure 11.1 depicts how we shift many of the traditional testing activities to the *left* on the timeline and automate them in **continuous integration and continuous deployment** (**CI/CD**) pipelines.

These include static analysis, unit testing, integration testing, contract testing, and transitive end-to-end testing. But they are all performed in isolation, with test doubles, to minimize the friction that increases lead time. We will cover these in detail in the *Continuous integration pipeline* section of this chapter.

It also shows how we shift other forms of testing to the *right* on the timeline—into production, where we monitor and alert on the signals that the system is emitting. These include automated smoke testing (that is, synthetics), which we execute on a continuous schedule, and anomaly detection, which alerts us to significant changes in **key performance indicators** (**KPIs**) so that we can focus on the **mean time to recovery** (**MTTR**) and fail forward quickly. We will cover these in the *Continuous deployment pipeline* section and dive deeper into them in *Chapter 12, Optimizing Observability*.

However, no matter how hard we try, testing cannot eliminate all errors—we need to focus on risk mitigation as well.

Focusing on risk mitigation

No process is perfect. We cannot eliminate honest human error. We can double down on our automated testing—even triple down, and more. But sooner or later, a mistake will happen, because to err is human.

The solution is not to slow down, but instead to go faster and *mitigate the risk*. We need to force ourselves to control the batch size, by decoupling deployments from releases, with the help of feature flags. And when things still go wrong—and they will—then we must be prepared to fail forward fast, while we rely on our autonomous services to limit the blast radius.

Small batch size

One of the most effective ways to mitigate risk is to control the batch size of the work units we produce. Agile methods help us mitigate the risk of building the wrong solution by delivering features more frequently. A smaller batch size allows us to elicit feedback from end users more quickly

so that we can make course corrections and eliminate wasted development effort.

But we can do better. We need to control more than just the batch size of a delivery (that is, a **release**). A release can easily contain hundreds of changes. When a mistake happens, how do we know which change is the culprit? Even before a release, how can we reason about the stability of so many changes? There is often an interplay between changes that is not readily apparent.

We need to minimize the number of changes, but we can only reduce the batch size of a release so much before it is no longer useful for eliciting end-user feedback. We need a work unit that is smaller than a release. We need a unit of work that helps teams maintain the stability of the system. We need to decouple deployment from release.

Decoupling deployment from release

The smallest possible batch size is a single change. If we deploy a single change and the system breaks, then the cause should be fairly obvious. More importantly, it is much easier to reason about the correctness of a single change—not necessarily that the change is functionally correct, but whether or not it will break the functionality that is already working.

End users do not like it when we break features that they rely on. They also don’t like downtime, which is equivalent to breaking a feature. Yet they always want more. We need to move fast, so we can innovate and give users more, but we certainly don’t want to break things. And as we will see in the *Achieving zero-downtime deployments* section, sometimes the only way to

accomplish this is by deploying the smallest possible backward-compatible change.

We need to decouple deployment from release (that is, delivery) and treat them as separate events. A **deployment** is just the act of deploying a change into an environment. We push the change all the way to production, and we do this many times a day without interrupting the end users. We will see how this works in the *Turning the crank* section, with automation pipelines, comprehensive testing, and appropriate gates.

The changes then sit dormant until we are ready to turn them on. We will enable them for ourselves so that we can perform controlled testing, but we wait until we have accumulated enough changes before we release the feature to the end users. At this point, the act of **releasing** is nothing more than flipping a switch to enable it for a set of users. These may be beta users, or we may roll out a feature progressively across business units. We may A/B-test variations of a feature with a percentage of customers to find the best fit.

Ultimately, we are continuously and progressively deploying small, controlled changes, many times a day, across many autonomous features. We will reach a point where we are continuously delivering new and improved features to various groups of users on a regular basis. We employ the help of feature flags to accomplish this.

Feature flags

Feature flags are not as fancy as they may sound. They allow us to deploy code to production without releasing it. A **feature flag** is just conditional logic that we add to the code to control when a feature is available and to

whom. But conditional logic can get messy. It increases the complexity of the code. It increases the magnitude of the testing effort.

If not well designed, a feature flag can do more harm than good. But feature flags are instrumental in mitigating the risks of continuous innovation, so it is worth the effort to properly design and implement feature flags. Let’s look at the difference between natural and artificial feature flags and a particular type of feature flag called keystones.

Natural

Natural feature flags are the best kind of feature flags. These are conditions that are intrinsic to the functionality—for example, most systems provide a role-based access mechanism that we use to control the rendering of menus and screens for authorized access. We can use this mechanism and craft a hierarchy of roles to control access to new features.

Some systems require the user to purchase a feature. We could use the pricing model to offer a feature for free to beta users for a period of time. Other features are naturally optional, with the ability to opt in and opt out. Alternatively, we can combine natural feature flags to achieve the desired result. In the *Achieving zero-downtime deployments* section, we will see how we can use the presence of a change itself as a feature flag.

Keystone

Our micro frontend architecture allows us to implement a natural feature flag that we call **keystone feature flags**. This flag gets its name from the wedge-shaped piece at the crown of an arch of a doorway or bridge that locks the other pieces in place. We can use a single keystone feature flag to control

access to a new micro-app, plus its BFF service, as well as any downstream functionality that consumes the events produced by the BFF service.

This is a time-saving technique. We can deploy a new service without an explicit feature flag, because if nothing ever calls it and nothing produces the events that it is listening for, then it is effectively feature flagged off. Then we use the keystone feature flag for the micro-app to enable all the related functionality with the single flag.

In *Chapter 10, Securing Autonomous Subsystems in Depth*, we saw how we can add metadata, such as roles, to mount points to control access to micro-apps. We can add additional metadata, such as `environment` and `region`, that will act as keystone feature flags. These allow us to deploy the micro-app and related services to all regions (such as east and west) in all environments (such as non-prod and prod) and selectively enable them as needed.

Artificial

We should always favor natural feature flags over artificial feature flags. An **artificial feature flag** has no intrinsic value to the application. It is Boolean logic that we temporarily wrap around a feature to limit access. It looks up the state of a flag and acts accordingly. After we release a feature, we need to remember to go back and remove an artificial feature flag.

Technical debt is the primary concern over feature flags. If we do not double back and remove these flags, the code can become unwieldy. Even natural feature flags may need cleanup, such as removing a role that we no longer need. We should specifically allocate cleanup tasks to the task roadmap so that we remove any technical debt as we proceed. We will see examples of this in the *Achieving zero-downtime deployments* section, and we cover task roadmaps in the *Planning at multiple levels* section. But try

and plan as we might, we must be ready to fail forward when things go wrong.

Fail forward fast

Again, no process is perfect. We cannot eliminate honest human error. We will make mistakes. So, we need to be prepared to jump into action at a moment's notice to fix the problem and move on.

Fortunately, the batch size of our deployments is now a single change. So, when something breaks, the cause is much clearer. We can easily triage the issue and define a path forward, not backward.

We are performing many deployments a day, and the process has become routine. We can rely on this **muscle memory** to roll out a fix quickly. The time needed to roll back the change is likely no different than the time to roll forward with the fix. So, we should favor the path forward.

Observability is crucial to minimizing the MTTR. We need to know that there is an issue before the end user does. We will see exactly where it fits into the process when we continue this journey in *Chapter 12*, *Optimizing Observability*.

Yet in the eyes of the end user, when something breaks, it is the equivalent of the system being down. If the user cannot use a feature, then the system is essentially offline, so our aim is always to have zero downtime. Let's look at what it takes to achieve zero-downtime deployments.

Achieving zero-downtime deployments

The days of taking a system offline for maintenance are long gone. Customers expect a system to be online at any time. Even employees, particularly regional employees, use systems around the clock. There is simply no viable window to take a system offline.

We need the ability to seamlessly perform deployments without interrupting the end users. This is one of the goals of **autonomous services** and **micro-apps**. Each piece is independently deployable and supports a different actor, which means that they deploy more quickly and impact different groups of users. Plus, in the event of a mistake, they limit the blast radius.

However, there are still dependencies between these components. For example, suppose we need to change the name of a field in the domain model. This change has a ripple effect. It impacts the interactions between the frontend and the backend, between the backend and the data store, and between the event producer and the downstream consumers.

We also need to avoid a versioning nightmare. We are performing many small-batch-size deployments, so running concurrent deployments is impractical. We will reserve versioning for significant changes in functionality, at the release level, which we will roll out to beta users, and so on.

Instead, we will leverage the small batch size of our deployments along with the *Robustness principle*.

The Robustness principle

The **Robustness principle** is a design guideline that helps us roll out incremental changes. It states the following:

Be conservative in what you send, be liberal in what you accept.

Let’s see how this can help us with an example scenario for changing a field name. The first part of the principle says: **be conservative in what you send**.

The backend needs to be conservative in what it sends to the frontend. If it sends a new field name to the frontend, how will the frontend react? Will the frontend ignore the field or throw an error? What if it removes a field? Will the frontend throw an error?

The backend needs to be conservative in what it sends to the data store. If it sends a new field name to the data store, how will the trigger function react? Will the trigger throw an error, or will it send out an invalid domain event without this data? How will downstream consumers react to events with missing information?

We need to avoid all these kinds of situations. They all constitute downtime. The second part of the principle says: **be liberal in what you accept**.

The frontend needs to be liberal in what it accepts from the backend. The backend can send both the old field and the new field, and the frontend can ignore the one it doesn’t expect; or the frontend can be modified to handle either field name and present a single field to the user.

The **trigger function** needs to be liberal in what it accepts from the data store. It can accept either field name from the data store and send both field names in the domain events. Consumers can accept either field and ignore unrecognized fields.

We can think of the field names as natural feature flags. The presence of one or the other field dictates how the code will behave. Then, we string together a series of deployments to make the necessary changes on both ends of an interaction, as follows:

1. The first deployment changes the backend to send the field with both the old and new field names.
2. The second deployment changes the frontend to use the new field name.
3. The third deployment changes the backend to stop sending the old field name.

These three deployments can proceed over a period of just hours, with zero downtime for the users. Only the third deployment requires some delay while we wait for the cache to expire on the user's browser.

These are rapid **one-two-three-punch** combinations of deployments that we can only achieve when we have *small batch sizes* and we *decoupled deployment from release*. We add a conservative change on one side, upgrade the other side, and then double back and remove the flag from the first side.

Conversely, if we were to deploy every 2 weeks, then this process would take 6 weeks. In other words, slowing down is simply not practical. Instead, reducing lead time, inverting our testing process, and focusing on risk mitigation opens the opportunity for new solutions to old problems.

Let's dig a little deeper into how this example scenario plays out between the different components.

Between the frontend and its backend

When we are planning out one of these three deployment cycles for a specific change, we need to choose a side to start on. For the interaction between the frontend and its backend, this will depend on whether we are using **GraphQL** or **REpresentational State Transfer** (**REST**).

For **GraphQL**, we need to start on the backend because the frontend specifies the field names it wants returned. So, the backend needs to understand the requested field names. We might proceed as follows:

1. Add the new field name to the schema and add resolvers to map the new field name to the old field in the data store.
2. Update the frontend to use the new field name.
    - Wait for the cache to expire.
3. Remove the old field name from the schema. Note that the data store is still using the old field name, so we will leave those resolvers in place a little longer.

For **REST**, we can start on either side, so we will start on the frontend side for contrast. We might proceed as follows:

1. Update the frontend to use either the old or new field name.
    - Wait for the cache to expire.
2. Update the backend to replace the old field name with the new field name.
3. Update the frontend to only use the new field name.

Now that we have taken care of the frontend and the immediate impact on the end user, we can move on to the downstream consumers.

Between producers and consumers

Again, we can choose whether to start on the consumer side or the producer side. However, we definitely cannot update the database first. Furthermore, we need to account for multiple consumers, and we may need to coordinate

with multiple teams, so the plan may drag out longer and it must account for the most vulnerable consumers.

First, before we add the new field to the domain event, we must ensure that downstream listener functions will not break. The **listener** functions should follow the **Robustness principle** and ignore fields they do not recognize. They should produce a **fault** event if required data is missing, but not if there is just additional information. Once all listeners have *liberal* mapping logic, we can proceed as follows:

1. Update the trigger function to include both the old and new fields in the domain event.
2. Downstream teams update their **listener** functions to support both fields. Preferably, they can do this ahead of the upstream change.
    - Wait for all downstream changes.
3. Update the trigger function to remove the old field name.
4. Optionally, downstream teams may remove support for the old field name from their **listener** functions. This will depend on how far back in time they need to support the replay of events from the event lake, because older events will only have the old field name. These will tend to be services that operate later in the *data life cycle*, such as analytics and archiving. To keep the logic clean, the teams can segregate this logic into different *pipelines* that filter on the event format.

External domain events and **External Service Gateway** (**ESG**) services play an important role here. Within an autonomous subsystem, the various teams will work closely together. This means that we can upgrade the downstream consumers within an autonomous subsystem relatively quickly, and we can remove the debt of the natural feature flags.

However, it will take external consumers longer to upgrade. This is where the *anti-corruption layer* of ESG services comes into play. The ESG services can hold the technical debt of the feature flags while we wait for the external services to upgrade. In the meantime, this allows the internal domain events, **Backend for Frontend** (**BFF**) services, and **control** services to stay clean and remove the feature flags more quickly.

Between the backend and its data store

The traditional approach for upgrading a data store involves running a conversion process to update all the records. However, the data store is the one piece that connects all the others, so it cannot change until all the other changes are in place. We also have many different databases that may need to change. Some of these data stores may be so large that a one-shot conversion is impractical. Other data stores may be lean and have a **time to live** (**TTL**) that removes older records, so there is little need for a conversion.

Furthermore, most of our serverless data stores are schemaless. So, we will apply the Robustness principle to the data layer as well. The backend will use the presence or absence of a field as a natural feature flag on a *row-by-row basis*. The **command** and **listener** functions will save records in the *new* format, and **query** and **trigger** functions will understand *both* the old and new formats.

The deployments for **query** and **trigger** functions will proceed as follows:

1. Expect some records to have the new field and others to have the old field, and map either field at the record level to both the old and new

fields at the interface level for a query response or a domain event.
    - Wait for the frontend and consumers to upgrade.
2. Continue to support the old and new fields at the record level, but only support the new field at the interface level.
    - Wait for all the records to convert and/or old records to expire.
3. Remove the record-level mappings for the old field name.

The deployments for **command** and **listener** functions will proceed as follows:

1. Map the incoming old field or new field to the new field at the record level and remove the old field from an existing record.
    - Wait for the frontend and producers to upgrade.
2. Remove the incoming old field mapping and continue to remove the old field from an existing record.
    - Wait for all the records to convert and/or old records to expire.
3. Remove the logic to remove the old field from an existing record.

For a large data store, it may make sense to have a conversion process as well. This process can execute concurrently with the changes made by the **command** and **listener** functions. For example, this process might process a month's worth of historical data each day until complete.

We can implement many of these various steps in parallel, but the order in which we deploy each step to production is the most important part of achieving zero downtime. This requires purposeful planning. Let's look at how we can plan our deployments and releases at multiple levels.

Planning at multiple levels

The process of **choreographing deployment and delivery** requires planning at multiple levels. We need to plan the deployments and the deliveries, and we need some planning in between. Exactly what we call these different levels isn't as important as having the rightsized units of work that enable teams to establish a workable cadence.

The following diagram illustrates the overall process. This diagram is similar to a traditional Agile snowman, but it is different in that we are continuously deploying changes and continuously delivering working functionality to the end users in production:

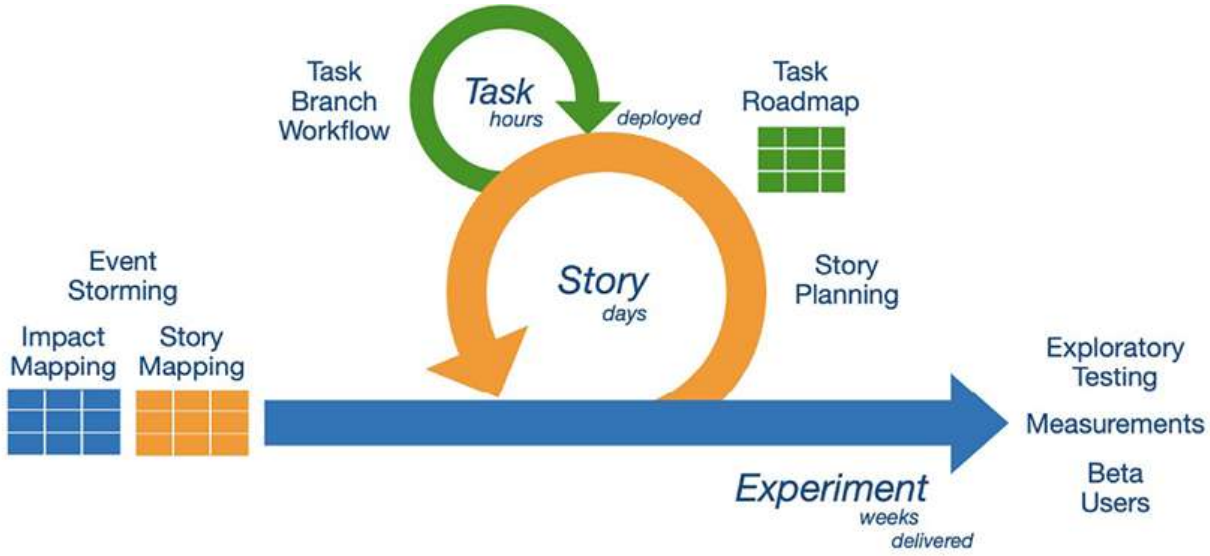


Figure 11.2: Multiple levels of planning

The process starts with some envisioning and event storming, as we covered in *Chapter 2*, *Defining Boundaries and Letting Go*, which produces an architecture of autonomous subsystems. From here, the individual subsystems can proceed, with relative autonomy and pertinent coordination, to formulate their plans. Then, the teams turn the crank, work the backlogs, evaluate feedback, and make adjustments.

For the purpose of discussion, we will call these levels of planning *experiments*, *stories*, and *tasks*. We will touch on some common alternatives to these names, but the three levels of granularity are what is most important to help control the batch size and mitigate risks.

Experiments

At the top level of planning, we focus on what we will deliver to the end user, and when. It is the largest unit of work, typically measured in **weeks**. A delivery equates to released software—that is, the ability to turn a feature on for a set of users. So, we can think of this level as *release* planning, but not exactly in the traditional sense.

Early deliverables will likely be internal releases, such as the *walking skeleton* that we will cover shortly, but we must deliver these to the *production* environment. Otherwise, we are accumulating a large batch size for the initial push to production that will interrupt the team's *rhythm* and increase the risk of a mistake.

We are ultimately delivering features to the end user, but we are delivering them incrementally. So, we can think of this level as *feature* planning, but it will likely take several *releases* (that is, experiments) before we are ready to turn a feature on for all users. In reality, we may have to adapt to early user feedback and change course several times before we have the right fit. So, at this level of planning, we will tend to call each unit an experiment.

We need to keep the batch size down, show progress, and receive feedback. It is also good for a team to establish a cadence and have regular milestones, but we need to deliver something meaningful without going off track. Thus, limiting the scope of an experiment to a few weeks, in the early phases, is a

good rule of thumb. As a product stabilizes, the experiments will likely be much smaller and more focused as we try to squeeze out more value.

Impact mapping is a common technique that teams use for planning at this level. It is *goal*-focused, such as we want to increase market share or improve efficiency by some percentage. Then, we formulate potential experiments to learn how we might achieve this impact for the specific users. The goal gives the team something measurable, so we can gauge whether an experiment is successful.

Epic is another common term in use at this level of planning. An epic is larger than a story and it might be smaller than a feature. It is probably larger than we want for an experiment. An epic may be a good unit of work for a modernization effort, where the team has experience with the functionality and a strong understanding of the scope of the work. However, the team should move to an experimenting mindset when they begin to introduce changes and new features.

From here, we divide experiments into stories that the individual teams divide into tasks.

Story backlog

There is plenty of discussion in the industry about the nature of **user stories**: what is a story, how do they differ from use cases, and so on.

Tactically, a **story** is just a *unit of work* that the team needs to perform to advance toward a delivery. We should measure these units of work in terms of **days**, to help control their scope so that the team can more easily reason about the impact the change will have on the production system.

We tell a story from the user's or actor's perspective. This is important because it aligns with our autonomous service patterns. Each service supports a specific actor, which makes it easy to *identify the impact* of the story.

We need to order the stories so that the team knows what to work on first, next, and so on. In addition, we need to split large stories into rightsized stories. Story mapping is a technique that helps us do this.

Story mapping

We use **impact mapping** to help steer the *direction* of the product, and **event storming** helps us uncover the *behavior* of the system. We also divide the system into autonomous subsystems so that each can change independently. Now, we need to divide the *delivery* of a subsystem into manageable units of work we call stories. This is where **story mapping** comes in. This is a technique that helps us order and rightsize stories on the backlog and craft releases. Jeff Patton defined the technique and coined the term in his article *The New User Story Backlog is a Map*: (https://jpattonassociates.com/the-new-backlog).

The following diagram depicts a story-mapping board:

Figure 11.3: Story mapping

We start by arranging user activities on the first row in the order that users perform them. Then, we add progressive details down the board for each user activity. Each card represents an ordered and rightsized story on the backlog. The team will pull the stories row by row, from left to right. Finally, we group rows into deliverables (that is, experiments).

The *first row* represents the first deliverable. It is the *walking skeleton* that forms the foundation for the subsystem. It is working software. The team identifies all the services and micro-apps, creates the repositories from templates, sets up the pipelines, and pushes everything through to production.

For example, each column aligns with a primary service. A column for a human actor will align with a micro-app and a BFF service, as we discussed in *Chapter 3*, *Taming the Presentation Tier*, and *Chapter 6*, *A Best Friend for the Frontend*. External activities will have their own columns and align with an ESG service, as we covered in *Chapter 7*, *Bridging Intersystem Gaps*.

And the flow of activities as a whole may align with a control service, as we covered in *Chapter 8, Reacting to Events with More Events*.

At this point, the structure of the subsystem is readily visible, with menus, navigation, and placeholders for the details. It is a rough sketch that the team demonstrates to the product owner to validate that things are off to a good start.

From here, the team iterates on the details of the user activities. Each story should be rightsized. If a story will take longer than a few days, then split it into multiple stories. For example, each section of a screen can be a separate story, and rules and validations can follow in additional stories. A **user-interface-first** (**UI-first**) approach may also make sense, with the early stories focused on the frontend and later stories implementing the backend.

It is important not to go too deep too soon. The early deliveries should target beta users and early adopters. Their feedback may change the direction, so we don’t want to waste effort on advanced details until we know we are on the right track.

Now that we have an ordered backlog (that is, a map) of stories, the team can start pulling stories.

Story planning

Story planning is the first activity the team performs when it pulls the next story from the backlog. This is akin to sprint planning, but slightly different. The focus is on a *single* story. The team has a conversation with the product owner to ensure they understand the intended functionality.

However, the most important output of the story-planning activity is a *task roadmap*. This roadmap enables the team to put the functionality into

production with *zero downtime* so that we can perform the experiment with real users and validate the product direction.

Task roadmaps

A **task** is the smallest unit of work. We should measure tasks in terms of **hours**. Each task is an atomic unit of work that only touches a single repository. A task is a *unit of deployment* and will update production once completed and approved.

Tasks on the same repository need to be serialized. Tasks for different repositories can proceed in parallel, but it may be necessary to serialize their deployment. We define this serialization in a task roadmap.

A **task roadmap** is a well-crafted series of tasks that enables the team to put the functionality of a story into production with zero downtime, as we covered in the *Achieving zero-downtime deployments* section. Each task deploys a backward-compatible portion of the functionality to production, following the *Robustness principle*.

Testing is not a task. Tasks are test-driven. A task includes writing the test cases to cover the new code. The overall test suite accumulates with each additional task.

Now let’s see how teams turn the crank to put the plan into action.

Turning the crank

Now that we have a plan, we get to do the fun part. We get to turn the crank and do the work. We pull the next *task*, write the code, push it to *production*, and see the fruits of our labor. Right?

Actually, up to this point, the process has been very flexible. It has been a lot of talk, sticky notes, and diagrams. It has been flexible on purpose—because we are experimenting.

But now, the rubber meets the road, so to speak, and we must get serious. After all, programming languages are notorious for doing exactly what we tell them to do, and we are about to deploy a change to production while the users are actually using the system.

So, this small part of the process needs *due diligence*. I call it the *task branch workflow*. This workflow *governs* how we turn the crank and do the work. It includes both automated and manual gates. But it is a straightforward workflow that becomes *second nature* since we do it many times a day, day after day.

And it is this **muscle memory**, which we have built up day after day, that we will rely on to jump into action and fail forward when we make an honest human error.

Task branch workflow

The **task branch workflow** is a GitOps-style continuous deployment approach that defines the machinery for moving *focused and deliberate* changes from development into production. It joins our issue trackers, developer tools, source code repositories, and CI/CD pipelines into a cohesive system that enables us to crank out working software. It is the *automated bureaucracy* that provides the needed guardrails so that we can work efficiently and safely.

The following diagram depicts the activities in a task branch workflow:

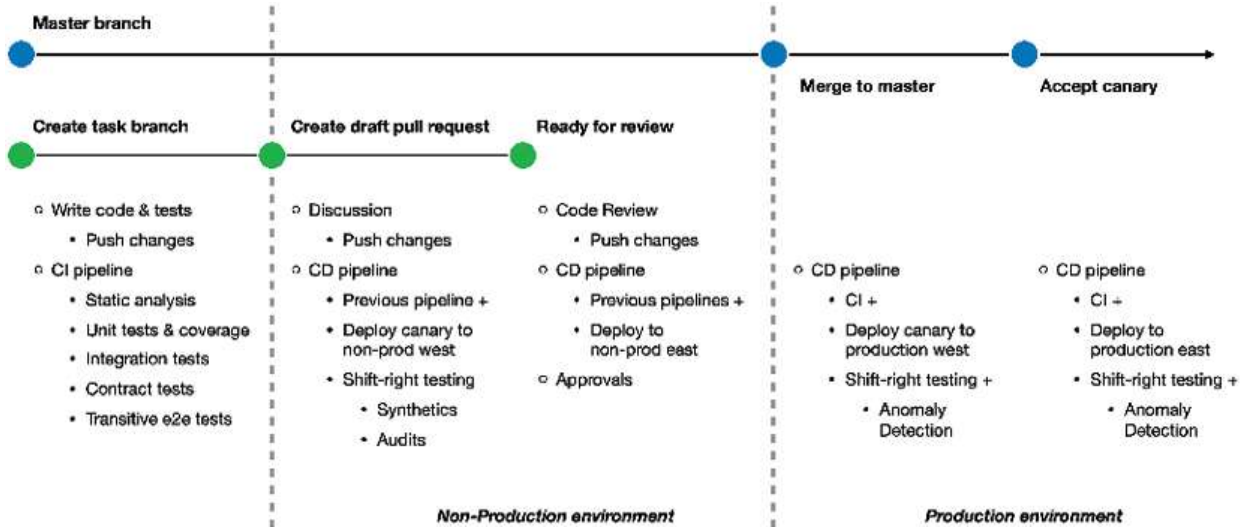


Figure 11.4: Task branch workflow

Let’s dig into the different activities of the workflow.

Create task branch

A task branch workflow begins when a developer pulls the next task from the backlog and creates a new branch off the master branch of the specific repository. We name the branch after the task, and we perform all work for the task on this branch. The branch will be short-lived, as we measure a task in hours.

The developer writes the new code and the all-important test cases for the new code.

We explicitly define the scope of each task when the team defines the *task roadmap*. Examples include seeding a new repository from a template, adding a new connector class, adding a new method to a model class, updating a mapper function for a stream processor, adding a new button to a screen, and so on. The key to success is that each change is focused and backward compatible. In other words, the change follows the *Robustness*

principle, as we covered in the *Achieving zero-downtime deployments* section.

The developer pushes the change to the remote branch, once all the tests pass. At this point, the CI pipeline will assert that all the tests pass. We will cover the details of the CI pipeline in the *Continuous integration pipeline* section.

Create draft pull request

The developer creates a draft **pull request** (**PR**) when the task is far enough along to warrant feedback and discussion. It is important to ask for feedback early and often. A good practice is to create the draft PR after the first push.

The small size and focus of a task branch are crucial to receiving prompt feedback from fellow developers, as they are working on their own tasks. They need to be able to zero in on the essence of the change, provide constructive feedback, and then get back to their own task.

Creating a draft PR also signals that it is time to deploy the change to the secondary/western region of the non-prod environment where continuous smoke tests are executing. We will cover regional canary deployments in the *Continuous deployment pipeline* section.

Ready for review

The developer moves the PR from draft mode into the ready-for-review status when the task is ready to go to production. The CI pipeline and the CD pipeline for the regional canary deployment to the non-prod environment must have already passed.

At this point, the developer opens the PR to a wider audience for code review and approval. How wide an audience and how many approvals will

depend on the risk level of the change. If the change is to a new feature or a rarely used feature, then there is low risk. However, if it is the most critical part of the system, then it will need multiple approvals and, most likely, outside approval. The team will identify the risk level during *story planning* and notify any outside approvers in advance.

A PR for a presentation-layer change should include annotated screenshots and/or video recordings. This opens the review process to the non-technical stakeholders and provides context for the technical review.

Moving the PR to the ready-for-review status also signals that it is time to deploy the change to the primary/eastern region of the non-prod environment where continuous smoke tests are executing. We will cover this in the *Continuous deployment pipeline* section.

Merge to master

The team merges the PR into the master branch only after all the automated and manual gates have passed. These include all required approvals and successful execution of the following:

1. The CI pipeline
2. The CD pipelines for all the regions of the non-prod environment
3. All the automated smoke tests in all the regions of the non-prod environment

Merging the PR into the master branch will trigger the execution of the CI pipeline against the master branch. Successful execution of the CI pipeline on the master branch triggers the deployment of the change to the secondary/western region of the production environment. We will cover regional canary deployments in the *Continuous deployment pipeline* section.

Merging to master is a manual gate. The team may merge straight away or wait for the optimal time. Again, the team will identify the need for any delay during story planning.

Accept canary deployment

Accepting the regional canary deployment is the final manual gate for a task-level change.

At this point, the team has deployed the change to one region in the production environment. If everything has gone according to plan, then any active users in this region are none the wiser, all the continuous smoke tests have succeeded, and there were no anomalies detected.

From here, the team will wait for a predetermined amount of time and then accept the regional canary deployment. This will trigger the deployment of the change to all remaining regions in the production environment.

If the team detects a problem that they did not catch in any of the regions of the non-prod environment, then they stop the line and jump into action. This is where they rely on their **muscle memory** to fail forward fast.

The high observability of the system and the limited scope of the change facilitate **root-cause analysis** (**RCA**). Meanwhile, the bulkheads of the feature limit the blast radius, so upstream and downstream features continue to operate unabated.

Now that we have asserted the stability of a deployment, it’s time to determine if we are delivering the right functionality.

Feature flipping

Decoupling deployment from delivery gives us flexibility—it allows us to divide a delivery into a series of zero-downtime deployments. The code is available in production, but it is inactive. We can selectively activate the code for exploratory testers and early adopters, and iterate on the feedback. Then, when we have the right functionality, we deliver (that is, **activate**) the feature for general availability. In other words, we have the flexibility to flip features on and off throughout the process of discovering the right fit.

Exploratory testing

This is where the more traditional **quality assurance** (**QA**) activities come into play. The testers manually exercise the system to determine if a feature is fit for purpose. Once again, the focus is on discovery. The testers may have notional test scripts for what they expect to find, but this exercise is really all about finding the unexpected. What we think is right and what works in practice are not always the same, and this is an opportunity to find the gaps.

Testers usually perform exploratory testing in the non-prod environment, but they may test in the production environment as well. In either case, we need to turn the feature on for testers. For example, granting the necessary permissions to the **Tester** role enables a new feature for all testers.

Beta users

Feedback from beta users is invaluable. These are real users, using a feature to do real work. They are early adopters and willing participants who are vested in the outcome, and they are willing to volunteer their efforts to make the product better. They also understand that the functionality is incomplete and that their results may vary as the team experiments.

Beta users perform their work in the production environment. Again, we make the feature available to beta users with a feature flag. For example, we may create a special beta user role, and there may be a dedicated registration or invitation process.

To win the support of beta users, it may be necessary to synchronize their work with a legacy version of the functionality. A beta user's time is valuable, and they do not want to have to do their work twice. They need the ability to switch between the old and the new. For example, they can start a task in the legacy version and complete it in the new version, and vice versa. If the new version is not working correctly, then they can switch back to the legacy version and pick up where they left off, to a reasonable extent. This does increase the level of effort, but it is a valuable risk-mitigation strategy and will likely reduce the level of effort in the long run. We covered legacy integration in *Chapter 7*, *Bridging Intersystem Gaps*.

General availability

In due time, we make a feature available to all users. The end product may barely resemble what we thought it would look like in the beginning, but that is OK. We needed to experiment to find the right solution for the end user.

We typically refer to this as a **General Availability** (**GA**) release. We might call exploratory testing an *alpha* release, and beta users use a *beta* release, but this time it is official, and with it comes a new level of responsibility.

We have turned the new feature on for all users, and it is time to remove the technical debt of any feature flags that we no longer need. For example, the standard user role now has all the necessary permissions, so we need to remove the specific tester and beta user roles. We also need to remove any

references to these roles from the code, along with any artificial-feature-flag logic, and push a cleanup deployment.

From here, any changes to a GA feature require additional scrutiny during story planning to ensure zero downtime. Any major changes to a GA feature may make it the new legacy and require concurrently deployed versions and data synchronization. But this is the kind of change and evolution we have designed our architecture to support, so bring it on.

Now, let’s look closer at the CI/CD pipelines.

Dissecting CI/CD pipelines

GitOps, and specifically our *task branch workflow*, is an event-driven paradigm that orchestrates the flow of code changes through our CI/CD pipelines to perform automated testing and automated deployments. As developers push code to a central Git repository, such as GitLab or GitHub, and interact with pull requests for code review, the central Git system produces events that trigger the execution of automated pipelines in a CI/CD tool, such as GitLab-CI/CD or GitHub Actions.

We refer to these pipelines as **Continuous Integration** (**CI**) and **Continuous Deployment** (**CD**) pipelines. Let’s look at each of these in turn.

Continuous integration pipeline

The **Continuous Integration** (**CI**) pipeline is the first gate in the task branch workflow. We cannot deploy the change to a non-prod environment, and we cannot merge the PR into the master branch until the code builds successfully and all the test cases pass.

The CI pipeline executes all the types of testing that we are automating and shifting left into the build process. These include static analysis, unit testing, integration testing, contract testing, and end-to-end testing.

Isolation from external dependencies is the key to executing all these test cases in the CI pipeline. Each pipeline must execute independently. We can achieve this more easily with autonomous services because they do not use synchronous inter-service communication. They only have a limited number of external dependencies, such as to the event hub and their data store.

In *Chapter 6, A Best Friend for the Frontend*, we highlighted the importance of wrapping external dependencies in *Connector* classes. These classes delineate all the locations where we need to create a *test double* to stand in for the real external service during testing, such as mock objects and VCR recordings.

It is also important for developers and the pipelines to execute the tests in the same way—for example, a developer can run all the unit tests and static analysis by executing `npm test` from the command line. Executing `npm run test:int` asserts the deployment configuration (that is, serverless.yml) and runs all the remaining test categories. These are the same commands that the pipeline executes. This allows developers to test their work before pushing a change, eliminates duplicate development effort, and decouples the test cases from the specific CI/CD pipeline tool.

The following example shows a fragment of a `.gitlab-ci.yml` file:

```
ci:
stage: test
script:
- npm ci
- npm test
- npm run test:int
```

The `npm` scripts do the real work of the CI pipeline. The CI/CD tool only provides orchestration and the execution environment. This example invokes the CI pipeline in the `test` stage on any branch.

You can find examples of pipelines for popular CI/CD tools here: https://github.com/jgilbert01/templates/tree/master/cicd.

Before we cover the different types of testing, we need to review some terminology.

Serverless testing honeycomb

Writing automated test cases is a time-consuming development activity. It is normal to have more test code than the actual code under test. This is certainly time well spent, but some types of test cases are harder to write and maintain than others. For example, end-to-end tests are the most difficult and unit tests are the least. So, we need a well-balanced portfolio of test cases.

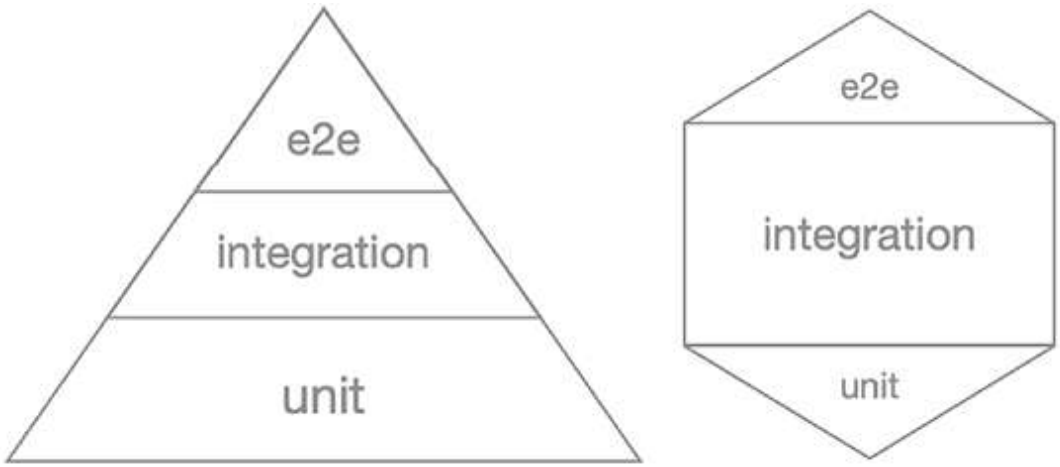


Figure 11.5: Test automation pyramid and testing honeycomb

Figure 11.5 depicts two opposing best practices, the *test automation pyramid* and the *testing honeycomb*. The more traditional *test automation pyramid*

recommends that we have more unit testing, whereas the newer, more microservices-focused *testing honeycomb* recommends more integration testing. Each has good reasons for its differing recommendations.

For example, the *pyramid* presumes that systems as a whole have more business logic that can be more easily tested in isolation using unit tests. This includes both *solitary* and *sociable* unit testing, as proscribed by Martin Fowler (https://martinfowler.com/bliki/UnitTest.html). Conversely, the *honeycomb* assumes that microservices have more integration points to test.

The problem with comparing these two models is that we are comparing apples and oranges. They each focus on a different scope or perspective. The *pyramid* takes a system perspective, and the *honeycomb* takes a service perspective. This brings us back to *hexagonal architecture*, which we covered in *Chapter 2, Defining Boundaries and Letting Go*.

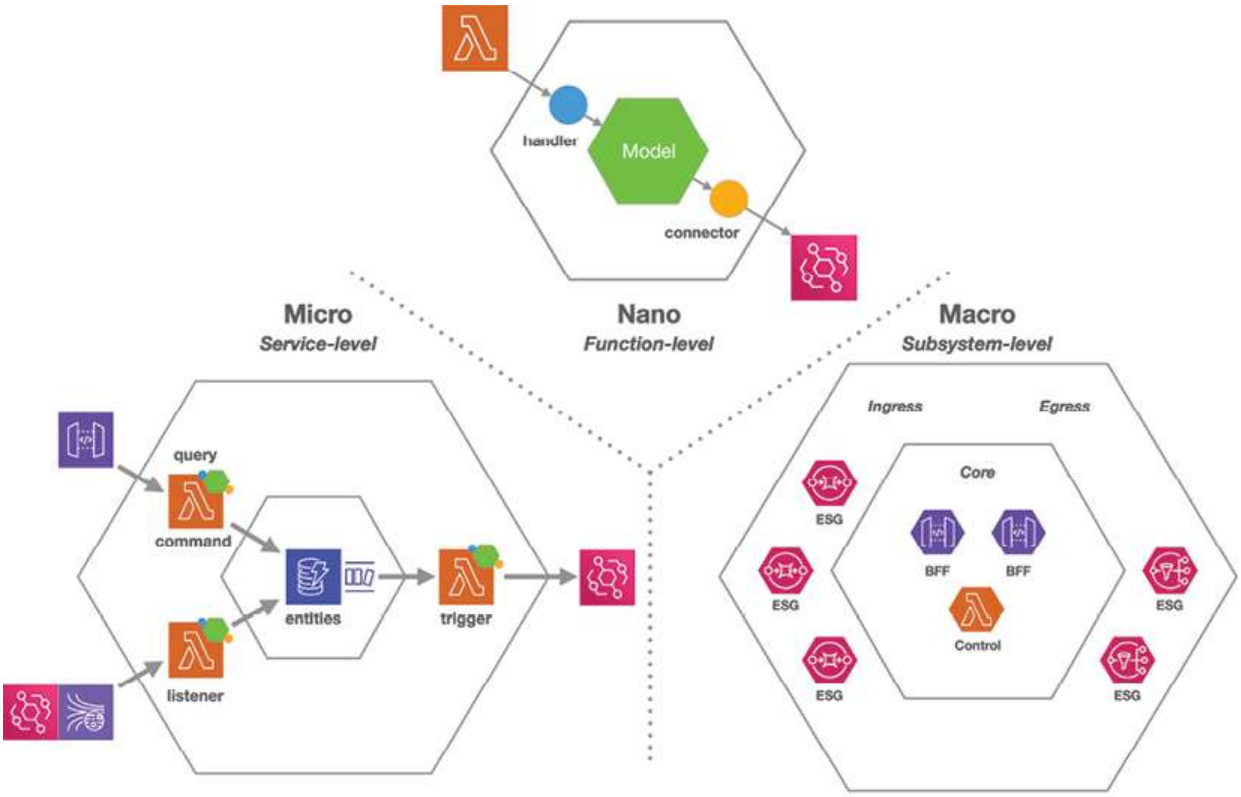


Figure 11.6: Hexagonal architecture

In *Figure 11.6* we see three different perspectives of our architecture: macro, micro, and nano. The *pyramid* model will usually hold true at the *subsystem level* (that is, macro), whereas the *honeycomb* model may hold true at the *service level* (that is, micro).

We also need to understand how the meaning of our testing terminology may differ depending on our context and perspective. For example, we can say that we have tested a given service from *end to end* and that we have tested a given subsystem from *end to end*. But those mean two different things and we may use different techniques to achieve those ends. The definition of *integration testing* can also differ depending on the unit under test and the ownership of its dependencies. Unfortunately, strive as we might for a ubiquitous language in our industry, we always need to make certain we check our terminology before a discussion goes off the rails.

When terminology is the culprit, it is often helpful to shift gears and address the problem from a different angle with a different but related set of terms. Fortunately, our serverless autonomous service patterns provide us with this opportunity, because we have the additional *function level* (that is, nano) in our hexagonal architecture with the terms *handler*, *model*, and *connector*. With this alternate terminology, I prefer the following **serverless testing honeycomb**:

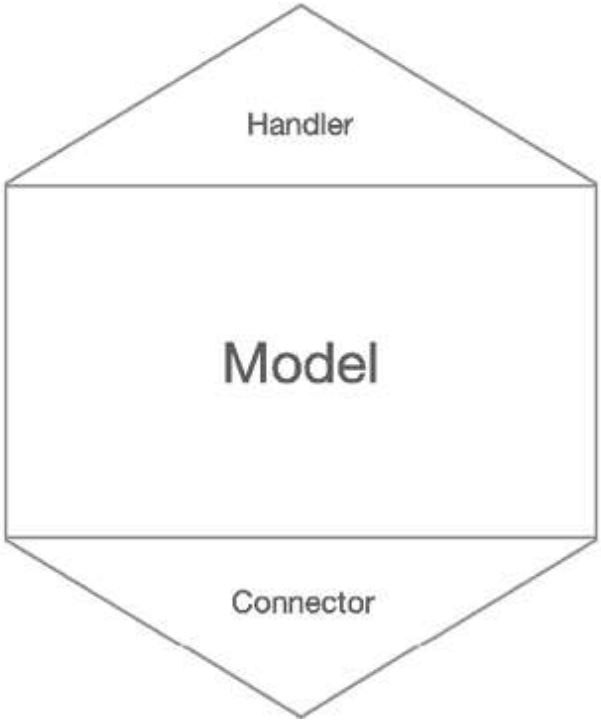


Figure 11.7: Serverless testing honeycomb

Figure 11.7 depicts the *serverless testing honeycomb* model. This model builds on the *ports and adapters* concept of hexagonal architecture and divides the testing effort into three different areas: **handler**, **model**, and **connector**. This honeycomb portrays the different kinds of code that we need to test. The handlers and connectors are adapter code, and the model is the business logic. We can test each with a different technique in a different

context. But by and large, we will have more test cases hitting the models than the handlers and connectors.

At the *function level*, we will perform *unit testing* with a focus on the models. We will use mock test doubles for the connectors and emulated events for the handlers. We will use a mixture of *solitary* and *sociable* testing. In particular, our stream processors lend themselves toward sociable unit testing.

At the *service level*, we will perform just enough *integration testing* with a focus on the handlers and connectors. We will use VCR test doubles for the connectors and the *serverless-offline* emulator to invoke the handlers. We will also craft these test suites so that the test cases cover the flow of the data through a service from *end to end*. We will also include some *contract testing* to ensure our models conform to the contract.

At the *subsystem level*, as the service contracts stabilize, we will perform targeted *end-to-end testing* with a focus on the data and events exchanged between services via the handlers and connectors. We will use a technique I call **transitive end-to-end testing**. We will also leverage strategic *synthetic* transactions to continuously test the system from *end to end* in non-prod and production.

Now let’s look into the different types of testing.

Unit testing

Unit testing is arguably the most important type of testing because it is the first line of defense against mistakes. We are continuously deploying small changes to the production environment, and unit testing is the best way to test that a change is not doing more harm than good.

Unit tests focus on testing the business logic within the individual components. For example, in *Chapter 6, A Best Friend for the Frontend*, we introduced the concept of *Model* and *Connector* classes. The bulk of unit-test cases will focus on the *Model* classes. These classes contain the domain logic and business rules. A fair number of test cases may be necessary to account for all the permutations in the logic. For rule-intensive logic, a portion of these tests will constitute the *acceptance tests* that assert the specified acceptance criteria.

Unit testing should constitute most test cases in a well-balanced *test automation pyramid* because they are the least difficult to write. To facilitate writing these tests, we test the business logic in isolation from their external dependencies. We accomplish this by substituting mock objects for the *Connector* classes, which we will cover in a moment.

You can find test case examples here: `https://github.com/jgilbert01/templates/tree/master/template-bff-service/test`.

Coverage

Unit tests are important enough that we fail the build if there is not 100% coverage. This may seem excessive at first, but with each task we only add a small amount of code, so we only need to write a small number of tests per task. Over time, these accumulate into an invaluable suite of test cases.

Without 100% coverage, it is too easy to fall into the trap of skipping these all-important tests. We are continuously deploying changes to production, so without proper testing we are putting the product, the process, and the mission at risk. We will make mistakes because we are human. The reputation of the product will suffer. The continuous deployment process will receive the blame for a lack of testing, and it will likely be discarded.

Ultimately, our ability to continuously innovate will be severely limited as we fall back on methods that are proven to increase lead time.

Code coverage doesn't help us write good tests, but it does keep us honest about how much testing we have and—more importantly—where it is missing. Of course, not every line of code is worth testing. So, we should make judicious use of `ignore` statements to keep coverage at 100%, without wasted effort. This way the coverage tool will tell us where we need test cases and keep us far from the trap.

Static analysis

The first step in the unit-testing process is **static analysis** (that is, **linting**). We perform this first because there is no point in unit-testing code that does not meet the guidelines.

Linting covers more than stylistic errors. It enforces best practices, automatically fixes many violations, and identifies common problems. For example, in React, we prefer functional components over class components, and linting can automatically convert class components to functional components. In essence, linting can help teach developers how to write better code.

The open source community has assembled many great rulesets, such as `eslint-config-airbnb`. Project templates should have these linting rules preconfigured so that each project gets off to the right start.

Mock objects

Testing should follow the scientific method, where we hold some variables constant, adjust the input, and measure the output. An important element of this is testing individual components in isolation from their external dependencies.

This is the primary goal of the *Connector* classes that we introduced in *Chapter 6, A Best Friend for the Frontend*. In the long term, connectors can facilitate changing an implementation decision, but in the short term, they provide immediate benefit by facilitating the testing of the business logic with mock objects.

A **mock object** is a form of *test double* that we use in unit testing to isolate external dependencies. They are pre-programmed by the developer with specific behavior to support a specific test case and substituted for the real object, following the substitution principle. The test case executes in isolation and asserts that the business logic invoked the mock object with an expected input.

However, we must handcraft the mock objects, which means they are prone to developer error, resulting in test cases that produce false positives. Furthermore, external interfaces tend to be verbose, which increases the likelihood of developer error.

Conversely, a *Connector* class provides a clean and simple interface to the business logic that encapsulates the details of interacting with the external resource. This greatly simplifies the work of creating mock objects for the majority of test cases and improves the reliability of the unit-test suite.

A small set of unit tests provide coverage for the *connector* classes themselves. For these tests, the developer spends extra effort to mock the more complex calls to the actual external interfaces. To account for developer error in these test cases, we will perform integration testing as well.

Integration testing

Integration testing picks up where the unit testing of connector classes left off. It ensures that the connectors are calling the remote services correctly.

Integration testing should constitute a small number of test cases in our *serverless testing honeycomb* and a well-balanced *test automation pyramid* because they are more difficult to write and maintain. This difficulty comes from the need to set up and configure the external dependencies prior to executing the tests. This includes initializing all the necessary data dependencies as well.

These calls also require the network, which is notoriously unreliable. This is the major cause of flaky tests that randomly and haphazardly fail. Flaky tests, in turn, are a major cause of poor team morale, since developers must waste valuable time tracking down the root cause of the test failures.

To solve these problems, we run the integration tests in isolation, as well, using a VCR test double.

VCR test double

We execute the integration tests in isolation by utilizing a test-double mechanism that intercepts the remote requests and replays recorded responses. We refer to this record-and-playback test double mechanism as a VCR after the video cassette recorder. Popular Node.js-based VCR tools include Replay and Polly.js. Let's look at an example of an integration test for a BFF service, as covered in *Chapter 6*, *A Best Friend for the Frontend*.

The following `serverless.yml` fragment configures the integration-testing environment. The `serverless-offline` plugin creates a local **HyperText Transfer Protocol** (**HTTP**) server that emulates the **Amazon Web Services** (**AWS**) API Gateway and Lambda runtimes. `baton-vcr-serverless-plugin` initializes the `Replay` VCR tool:

```
plugins:
...
- baton-vcr-serverless-plugin
- serverless-offline
```

The `npm run test:int` script starts the emulator and then executes all the integration tests. The following code block shows the integration test case for the BFF service's RESTful interface:

```
const client =
  require('supertest')('http://localhost:3001');
const THING = {...};
describe('bff command and query test', () => {
  it('should save', () =>
    client.put('/things/1').send(THING)
    .expect(200);
  it('should get by id', () =>
    client.get('/things/1').expect(200)
    .expect((res) => {
      expect(JSON.parse(res.text))
        .to.deep.equal(THING);
    }));
});
```

The test suite submits HTTP requests to the local emulator running on `localhost:3001`. The first test case saves a domain entity, and the second test case retrieves the domain entity by its identifier. The BFF service should execute the update and query operations against the DynamoDB table and return a response. The test cases assert that they received the expected responses.

> **NOTE**

I prefer to use a lightweight emulator, like the serverless-offline plugin, versus a more expansive emulator, like LocalStack, because there are fewer headaches that eat up time and effort.

There are no added installation steps or external prerequisites.

It only emulates the more straightforward interaction with the handler layer.

The connector layer invokes the real resources in the non-prod secondary/western region, so there are no emulation inaccuracies. We seed the stack from a project template with the typical configurations and permissions, as the first task in the roadmap.

Equally important (if not more), calling the resources in this environment forces us to think about the Robustness principle and zero-downtime deployment.

When creating a test case, the developer executes the tests in *recording* mode to create the recordings. The VCR tool intercepts the calls to DynamoDB and saves the request-and-response pairs to the `fixtures` folder.

The integration test suite is self-initializing—in other words, it creates all the necessary test data. This simple test suite initializes the data by executing the `save` command before executing the query. A more complex suite would set up reference data and related transactions in a before-suite block. The VCR records these initialization commands as well.

With the recordings in place, the CI pipeline does not need access to external resources. The pipeline executes the tests in *playback* mode. The VCR tool intercepts the calls to DynamoDB, looks for the matching recordings, and returns the recorded responses. If the code produces a different request or the VCR returns an unexpected response, then the test case will fail.

Integration testing also asserts the handler layer. This example demonstrates how the `serverless-offline` plugin emulates the AWS API Gateway environment for `command` and `query` functions. For `listener` and `trigger` functions, developers take events from the log files and use them to drive the test cases.

Integration tests should not cover all the functional scenarios. That is the job of unit testing. We need just enough integration testing to be confident that the handler and connector layers are processing and producing messages properly.

From here, we need to guard against the service side making changes that invalidate the recordings. That is the job of contract testing.

Contract testing

Contract testing and integration testing are two sides of the same coin. Integration tests ensure that a consumer is calling a service correctly, whereas contract tests ensure that a service continues to meet its obligations to a consumer and that any changes are backward-compatible.

Contract testing is unique because it is consumer-driven. This means that the consumer team writes a test case and submits a PR to the service team to add the contract test to its repository. The service team is not supposed to modify these tests.

If a contract test breaks, this indicates that the service changed in a backward-incompatible way. The service team must make the change compatible, following the Robustness principle, and then work with the consumer team to create an upgrade roadmap.

Backward-incompatible changes can manifest in subtle ways that may not violate a handcrafted contract test. For this reason, we will use the same recording from the consumer's integration test to drive the contract test in the service's test suite.

How we implement a contract test varies based on whether it is a synchronous or an asynchronous contract. Let's look at the testing differences between these contract styles.

Synchronous contracts

A synchronous contract between a frontend and its BFF service provides a prototypical example of consumer-driven contract testing. Although a single team owns both sides of this interaction, it is still worthwhile to write contract tests. The frontend and backend have independent pipelines, so there is a chance for incompatible changes to slip through.

First, the team creates an integration test in the frontend project that records the call to the backend. Then, they copy the recording to the backend project and use it to drive a contract test. The tests on each side ensure that the code is not changing in relation to the request-and-response recordings. If these tests fail, then the team must plan out a backward-compatible roadmap for rolling out the change.

The following code block shows the contract test case for the backend:

```
const relay = require('baton-request-relay');
const client =
```

```
    require('supertest')('http://localhost:3001');
describe('contract/frontend', () => {
  it('should exec the frontend get request', () => {
    const rec = relay('./fixtures/frontend/get');
    return client[rec.request.method](rec.request.path)
      .set(rec.request.headers)
      .expect(rec.response.statusCode)
      .expect(rec.response.body);
  };
});
```

The contract test uses the same emulator environment and VCR tools that we covered in the *Integration testing* section. The test case uses the `baton-request-relay` module to parse the recording. Then, it uses the recording to execute the same request as the consumer and compares the actual response to the expected response.

Asynchronous contracts

For an asynchronous contract, we need to invert the process for creating and recording a consumer-driven contract test because the consumer does not initiate the interaction. The upstream service produces the domain event, and zero-to-many downstream services consume the domain event.

There is no explicit coupling between the upstream service and the downstream services, yet the definition of the domain event forms the contract that the producer must uphold. So, an asynchronous contract test ensures that the upstream service continues to produce a domain event in the expected format. It also helps to document all of the downstream dependencies.

First, a consuming team creates a contract test in the upstream project that produces and records the expected domain event. In the following example, the test case asserts that the `trigger` function continues to produce the

recorded domain event for the given `CDC_EVENT` event. This test prevents the upstream team from making changes that will break the downstream service:

```
import CDC_EVENT from '../../../fixtures/downstream-consumer-x';
import { handle } from '../../../src/trigger';
describe('contract/downstream-consumer-x', () => {
  it('should publish thing-created', (done) => {
    handle(CDC_EVENT, {}, done);
  });
});
```

Then, the consuming team copies the recording of the domain event to their own project and uses it to drive a `listener` test. In the following example, the test case asserts that the downstream `listener` function understands the format of the recorded `thing-created` domain event. The test case uses the `baton-event-relay` module to parse the domain event from the recording:

```
import relay from 'baton-event-relay';
import { handle } from '../../../src/listener';
describe('contract/upstream-provider-y', () => {
  it('should process the thing-created event', (done) => {
    const rec = relay('./fixtures/upstream-provider-y/thing-crea
    handle(rec.event, {}, done);
  });
});
```

The various teams independently create integration and contract tests as they work on their stories and tasks. When the functionality of the system begins to stabilize, the teams can coordinate and leverage these testing techniques to create a cohesive end-to-end suite. We will cover this next.

Transitive end-to-end testing

Traditional end-to-end testing is time-consuming and expensive. It favors large batch sizes and long lead times, which is the exact opposite of what we are aiming for. All too often, teams skip end-to-end testing entirely and hope for the best.

Now, we have a better option. We can leverage the integration and contract testing techniques and craft a series of test cases that form a cohesive end-to-end scenario across all the projects (that is, **services**). Each contract test case produces its own set of integration recordings, and we use these recordings to drive the next test case in the next project, and so forth.

Figure 11.8 illustrates how these tests fit together. Borrowing from the transitive property of equality, if Test Case **1** for Service **A** produces Recording **X**, which drives Test Case **2** for Service **B** to produce Recording **Y**, which drives Test Case **3** for Service **C** to produce Recording **Z**, then we can assert that when Service **A** produces Recording **X**, then ultimately Service **C** will produce Recording **Z**:

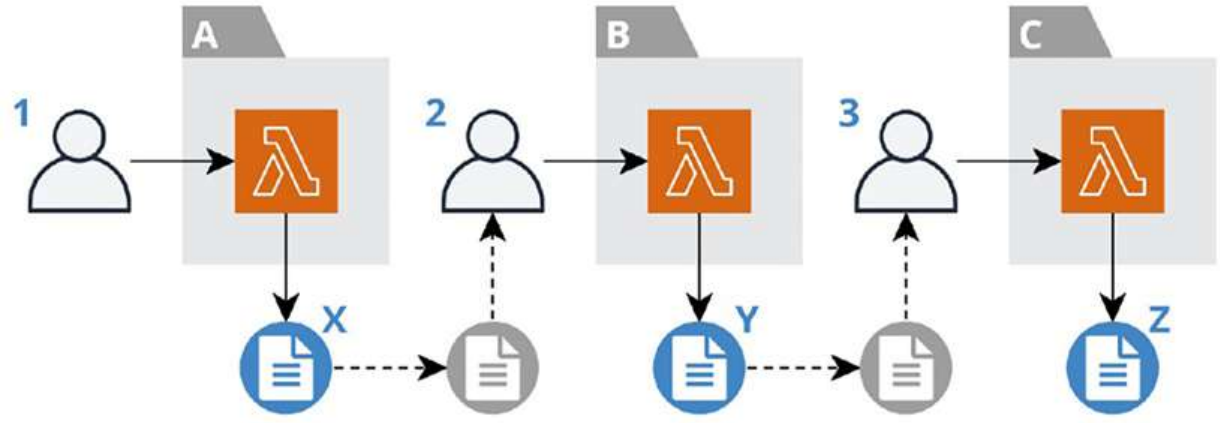


Figure 11.8: Transitive end-to-end testing

It is important to point out that the pipelines of the individual services remain completely independent. We only need to execute the portion of the end-to-end test suite that applies to a specific service when we modify it.

There is no need to rerun all the test cases across all the projects. So long as the recordings still match, we can transitively assert the correctness of the scenario. Only a backward-incompatible change requires dependent projects to rerun their portion of the end-to-end test suite.

Transitive end-to-end testing is a powerful approach that makes end-to-end testing practical because it significantly reduces the effort of maintaining an end-to-end test suite. There is no need to maintain a dedicated testing environment because the individual pipelines constitute a virtual environment. There is no need to maintain database scripts to reset databases to a known state because the recordings embody the data.

The test engineers from across the projects must collaborate to craft the end-to-end scenarios, and the developers have to implement the test cases and generate the recordings. But this is effort well spent. We leverage it over and over with each execution of a pipeline.

It is important to point out that the effort to put an end-to-end test suite together is a unit of work in and of itself. The teams will usually collaborate on this test suite toward the end of an experiment, once the functionality starts to stabilize. With each experiment, the teams will refine the end-to-end test suite.

Once an end-to-end suite is in place, the teams must keep the suite up to date as they work on each additional story and task. Therefore, it is important to keep the scope of the end-to-end test suite in check. It should only form the tip of a well-balanced test automation pyramid. The suite should have enough coverage to give the teams the confidence to continuously deploy changes, but not be so large that maintaining it impedes experimentation and innovation.

The CI pipeline gives us enough confidence to deploy a change, but we will proceed with several additional controls in the CD pipeline.

Continuous deployment pipeline

The **Continuous Deployment** (**CD**) pipeline executes the fully automated deployment of an autonomous service or micro-app to the non-prod and production environments. Once again, the CI/CD tool only provides orchestration and the execution environment. The pipeline delegates the heavy lifting to the `npm` scripts. The following example shows a fragment of a `.github/workflow/cd-np.yml` file:

```
on:
pull_request:
branches: [ master ]
jobs:
deploy-non-prod-west-canary:
steps:
...
- run: npm run dp:np:w
deploy-non-prod-east:
if: github.event.pull_request.draft == false
steps:
...
- run: npm run dp:np:e
```

This file defines the CD pipeline for the non-prod environment. The pipeline executes when the developer creates a PR that targets the master branch, and on any push thereafter. The *canary* deployment to the secondary/western region has no additional guard conditions, whereas the deployment to the primary/eastern region asserts that the PR is not in draft mode before proceeding.

The following example shows one of the `npm` scripts that the pipeline executes:

```
  "dp:np:e": "sls deploy --region us-east-1 --stage np"
```

The script calls the Serverless Framework, which in turn delegates to AWS CloudFormation to deploy all the resources of the service. You can find examples of `package.json` and `serverless.yml` files and pipeline files for popular CI/CD tools here: https://github.com/jgilbert01/templates.

These fully automated and declarative deployments go a long way toward eliminating human error. But before we go all-in on the deployment, we have one more gate. We will test the water first with a regional canary deployment.

Regional canary deployments

The phrase *canary in a coal mine* refers to the former practice of taking a caged canary into a coal mine to alert miners of the presence of deadly gases. A canary is more susceptible to toxic fumes and would provide an early warning sign when it stopped singing.

In software engineering, a *canary deployment* is the practice of rolling out a change to a subset of users, before rolling it out to all the users. The subset of users acts as the canary. If there is a problem with the change then it only impacts the subset of users, and the team fails forward with a fix; otherwise, the team increments the percentage of users, such as 1%, 10%, 30%, 70%, 100%.

However, many—if not most—services will not benefit from traditional canary deployments because they do not have enough traffic; for example, a service may only receive sporadic usage. Yet, we still want the benefits of incremental deployment.

We also need to limit the risk of a deployment to an isolated environment—for example, we should not deploy a canary into the same cluster that is running the previous version of the code. The level of risk may seem low, but the consequences are significant if a mistake takes down the entire cluster.

This is where multi-regional deployments provide significant benefits, because each region is an isolated environment. Whether they are running *active/active* or *primary/hot-secondary*, we can leverage these isolated environments to perform regional canary deployments, and we will rely on the failover capabilities that we already have in place, which we covered in *Chapter 9, Running in Multiple Regions*. We need to have multiple regions anyway, so let's get the most out of them.

If a service is running in a *primary/hot-secondary* configuration, then this approach is akin to a *blue-green deployment*. We treat the secondary region as the green environment and the primary region as the blue environment. The first deployment goes to the green region. If the smoke tests succeed in the green region, then we move ahead with the deployment and smoke tests in the blue region.

If a service is running in an *active/active* configuration, then we pick a region—usually a western region—as the canary region. The subset of users in the region acts as the canary. The first deployment goes to the canary region. We also execute the smoke tests against the canary region, since the user activity is sporadic. If the service does have a significant number of

active users, then we can add weighted routing to the equations to achieve a more traditional canary deployment.

If there is a problem with the regional canary deployment, then we rely on the failover capability and stop routing traffic to it. Once we are satisfied with the deployment in the canary region, then we move ahead with the deployment and smoke tests in the remaining regions.

We apply this approach to the non-prod environment as well, for an added level of protection. In the non-prod environment, this will tend to have a more blue-green feel, because there are far fewer testers using this environment as compared to active users in production. Regardless, we need to treat non-prod with the same mindset as production so that we can verify the zero-downtime task roadmap and practice the incremental deployment process.

Regional canary deployments give us the benefits of both isolation and incremental deployment, regardless of the level of user activity, on a service-by-service basis. Now, let’s look at simulating user activity so that we are continuously exercising the system with smoke tests.

Synthetics

Our *shift-left testing* provides a significant level of protection because these tests must pass before we can deploy a change. However, a CI pipeline only executes the tests for the repository that is changing. Plus, we are using test doubles for all the dependencies.

The contract and transitive end-to-end tests provide strong backward-compatibility protections for the upstream and downstream services. However, there is always the potential that we might miss something—for

example, a change may introduce a subtle dependency between components that we didn’t intend to occur.

This is where *shift-right testing* and *synthetics* come in. These are akin to traditional end-to-end testing, but we execute them against the non-prod and production environments. **Synthetics** are essentially **smoke tests** that we automate and execute continuously.

These tests form the top of a well-balanced test automation pyramid. They cover the most critical functionality of the system. This is the 20% of the system that we use 80% of the time. They cover the functionality that must work, or else. Of course, all functionality is important, but we cannot test the entire system continuously. So, we have to make decisions.

Continuous means that we execute these tests on a schedule—for example, every 15 minutes to once an hour for the most important tests, or once a day or once a week for features that are equally important but used far less often.

We execute these tests in the non-prod environment as well as the production environment so that we can find issues before approving a PR, and we execute them independently in each region to assert the stability of the canary deployments. For a regional canary deployment of a service with light usage, the synthetics will act as the **canary** users.

After a deployment, the team must pay extra attention to these continuous smoke tests to confirm the health and stability of the system. The bottom line is that the team needs to know about a problem before the end users so that they can fail forward fast.

Even when we are not changing the system, these tests help assert the overall health of the system. For example, if the cloud provider experiences a regional disruption, the synthetics will help trigger a regional failover.

When a synthetic test fails, then it is obvious that the team must jump into action. However, there may be an indirect problem downstream. This is where observability and anomaly detection come into play. Synthetics continuously exercise the system, and anomaly detection continuously tests the output signals. Together, they continuously check the health of the system so that the team can focus on their experiments. We will continue this journey in *Chapter 12*, *Optimizing Observability*.

Summary

In this chapter, you learned why it is important to control the batch size of a deployment and how to decouple a deployment from a release with feature flags. You learned about the Robustness principle, how to leverage it to achieve zero-downtime deployments, and the role that a well-planned task roadmap plays in the overall process.

We dug into the details of the task branch workflow and CI/CD pipelines, and you learned about regional canary deployments and synthetic smoke tests. You also learned how to implement continuous testing that enables continuous deployment and continuous discovery, as the team iterates to the right product fit.

In the next chapter, we will build on our continuous deployment pipelines and round out our testing strategy with observability. We will collect resource metrics and shift some testing to the right into production, where we will alert on anomalies in our work metrics. We will also cover FinOps and continuous tuning.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/B8TubSxc35

12

Optimizing Observability

In *Chapter 11*, *Choreographing Deployment and Delivery*, we laid out a lightweight development process that helps us *decouple deployment from release* and turn the crank with a *task branch workflow*. In addition, we defined an approach for continuous testing that allows us to *discover* the requirements and continuously deliver business value.

Throughout this book, we have focused on creating **bulkheads** within our systems to control the blast radius when things go wrong. Now we pick up where we left off in our continuous deployment process. We need to **monitor** the behavior of the system in production so that teams know when something is wrong and provide them with the information they need to **fail forward fast**.

In this chapter, we're going to cover the following main topics:

- Failing forward fast
- Turning observability inside out
- Leveraging FinOps
- Collecting resource metrics
- Tracking system events
- Alerting on work metrics
- Observing real user activity
- Tuning continuously

Failing forward fast

Observability is a multi-faceted topic. It can mean different things to different teams. Traditionally, observability has focused on monitoring infrastructure. However, with our serverless systems, we are delegating that responsibility to the cloud provider. This means that we can use observability at a higher order and apply it to help us drive innovation and deliver business value. In other words, **observability can help us drive down lead time and move faster**.

In *Chapter 1*, *Architecting for Innovation*, we dissected lead time so that we can understand what causes it to increase. For example, teams will naturally put on the breaks and slow down when they fear that a change could inadvertently break another part of the system. This is why we build *bulkheads* throughout our systems.

Teams will also slow down when they do not have enough information about the health and performance of the system. No process is perfect. We cannot eliminate honest human error. Teams will make mistakes. Therefore, when a team pushes a change to production, they need to be confident that they can **find and fix problems fast**. Otherwise, they will be apprehensive about moving forward and innovating.

This is where observability comes into play at the application level. We need to put working software in the end users' hands in production so that we can get the feedback we need to discover the real requirements. If a deployment has an unintended side-effect, then we want the system to **alert the team**, preferably before the end user does. Then we can **jump into action**, at a moment's notice, and use the detailed observability information to find the root cause and minimize the **mean time to recovery** (**MTTR**). In other words, **observability helps us fail forward fast**.

Now, let’s see how serverless makes observability easier.

Turning observability inside out

I remember when a product owner asked me, many years ago, why her product had periodic performance issues. I didn’t have a good answer because in those days it was typical to only monitor the infrastructure, with details limited to the likes of CPU and memory utilization and overall transaction volumes and latency. I could confirm that there was a periodic issue, but there was very little contextual information available to identify the root cause.

The system was also a monolith, which made it difficult to observe one transaction type from another. So, I went about instrumenting the code. I used aspect-orient programming techniques to inject code between the various layers to collect and tag detailed metrics. It took a fair amount of elbow grease, but eventually, I had the information that I needed to find the root cause of various problems and improve the performance of the system.

Today, serverless turns this problem inside out. Each serverless resource we use, such as functions, data stores, and messaging channels, provides us with detailed and tagged metrics by default. Our design patterns group these resources into autonomous services so that we can correlate related metrics and see exactly how individual transactions are performing. We can also leverage the different function types, such as listener, trigger, command, and query, to analyze performance across services.

We have tons of data about the inner workings of our serverless systems right out of the box. Instead of spending valuable time instrumenting our

code, we can get straight to the important part, which is turning all this fine-grained data into actionable information that autonomous teams can use to **fail forward fast**. We will learn how to do this in this chapter by categorizing this data into **resource metrics** and **work metrics** and tracking **system events**.

But first, let's look at how we can leverage cost as a useful metric for serverless systems.

Leveraging FinOps

Cost is always a primary concern, particularly when teams are first venturing into the serverless arena. But serverless not only turns observability inside out; it turns the concern for costs on its head. Instead of worrying about costs, we will leverage them as a powerful tool for self-governance, because the correlation between serverless costs and performance enables autonomous teams to utilize this **financial** information to continuously tune and optimize the **operation** of their subsystems.

Let's see how we can use cost as a metric, dissect the typical monthly serverless cloud invoice, and explore worth-based development and tuning.

Cost is a metric

Cost was not a very useful metric back when I first started building and running systems in the cloud, well before serverless came around. In those days my monthly cloud bill just told me how many hours of AWS EC2 instances and AWS RDS databases I had used. Those values were very monolithic. They did not tell me how I was using those instances and

databases. They did not tell me how well I was utilizing those resources. And they certainly did not tell me whether I was getting my money's worth.

Once again, serverless turns this problem inside out, just like observability. We can see in much greater detail where our money is going. We can see how much we are spending on each serverless cloud service we are using, such as functions, different messaging channels, and various datastores.

We can see how many times we invoked each cloud service. For messaging channels and datastores, we can usually see how many of those invocations were reads versus writes. Plus, we can see exactly how much time we spent on those invocations.

Nowadays, cost is a very useful and powerful metric. You could even say that cost metrics are empowering because instead of deflating team morale, they provide us with valuable information that we can use to improve our systems. We start thinking about creative ways to optimize the number of transactions and reduce transaction times. For example, using low-cost edge caching reduces the number of requests not just to the API gateway, but all the way through the stack down to the datastore. Plus, optimizing an algorithm to reduce transaction time not only lowers cost but also improves performance and user satisfaction.

This new reality is a bit mind-bending because we traditionally need to spend more money to improve performance. Even when we optimized an algorithm under the traditional approach, this usually just meant that we got more headroom within the capacity we had already purchased. This changes the incentive equation. I might even say that performance tuning becomes addictive. Each month, when the cloud invoice arrives, the cost metrics act as a powerful motivator for self-governance and continuous improvement.

Dissecting monthly cloud cost

Without fail, every month that you use the cloud you will get a monthly invoice. Also, without fail, your monthly bill will be full of surprises, some good and some not so good. Either way, they are full of useful information. I have run serverless systems for many organizations including Fortune 500 companies, government agencies, and startups. These systems follow the patterns in this book and process many millions of events per hour. The overall costs do vary by the size of the business, but they all generally take the same shape.

I like to group the various costs on a cloud invoice into three categories: **expected cost**, **unexpected or hidden cost**, and **governance cost**. Surprisingly, in my experience, each of these categories tends to account for a third of the total cost. This is surprising because two-thirds of the cost are not even on most company's radar when they perform their estimates. Let's look at these categories in reverse order.

Governance costs include things like continuous auditing and backups, which we have covered in *Chapter 10*, *Securing Autonomous Subsystems in Depth*, and observability, which we are covering in this chapter. They are often a surprise because a central group usually controls these and automatically provisions them to each account. This group may not be paying close attention to the cost of each account so you may need to do so. We can think of these costs as an insurance policy. If you do not have any of these costs, then your system is likely at risk.

The **unexpected or hidden costs** include things like logging, encryption, replication, and data transfer costs, plus taxes. These are usually a surprise because they are harder to predict, and they can fly under the radar until we

are running at scale. The good news is that these costs are within our control. We can tune these costs with techniques like caching and compression. For example, our stream processors can generate a lot of logging statements, but instead, we can collect this data in a *unit of work* and only log it when there is an error or based on a sampling rate. Regardless, we can think of these costs as an opportunity to improve our systems.

The **expected costs** include the various serverless cloud services that we are using to implement our autonomous services, such as functions, the different messaging channels, and the various datastores. The thing that may be surprising is that the compute cost (that is, functions) tends to be a fraction of the data cost (that is, datastores). The cost of serverless does not tend to live where everyone expects it to live. Most of the cost revolves around our serverless datastores, which is a good thing considering our data is our most important asset.

But regardless of where the costs live, the **Total Cost of Ownership** (**TCO**) for our serverless-first systems tends to be lower than non-serverless systems, largely because our costs scale to zero in our lower environments. Plus, we have a lot of information that will help us determine if it is worthwhile to optimize the cost and performance. So let's see what this is worth.

Worth-based development

I first ran across the concept of *worth-based development* in an article by Simon Wardley called *"Why the fuss about serverless?"* (https://www.linkedin.com/pulse/why-fuss-serverless-simon-wardley). These ideas are a key part of what

FinOps is today. The basic idea is that we can use cost metrics along with other observability metrics to decide whether it is worth spending our time and effort on tuning and optimizing our autonomous services.

For example, let's say that a specific transaction within an autonomous subsystem account for 60% of the monthly cloud cost. If the team determines that they could improve the performance of that transaction by 50%, then that would account for a 30% reduction in monthly cost. At this point, it becomes a simple matter of deciding whether the estimated cost of the improvement is worth the expected reduction in monthly runtime cost. Even if the cost is equal in the short term, the improvement in customer satisfaction and goodwill may be well worth the effort.

I usually take the larger line items on the bill and use other metrics to determine which autonomous services are most responsible for those costs. Then I zero in on those services to look for potential optimizations. Sometimes you find a quick win that takes a big bite out of the monthly cost, but usually, the simple changes come in small bites that can add up over time.

Many improvements require more elbow grease and may not be worth the investment now. Nevertheless, serverless will provide us with the data we need to make informed decisions.

Now, let's see how we can collect the resource metrics that we will need for root cause analysis.

Collecting resource metrics

So far, we have seen that serverless and autonomous service patterns provide us with a lot of fine-grained metrics without the need for explicit

instrumentation. Now we need to turn all this data into **actionable information** so that we can fail forward fast.

Back in 2015, when I first found myself in the role of operation manager, I went looking for guidance so that I could put together a mental model for reasoning about all this data. One of the best sources I found was a series of posts by Alexis Lê-Quôc, Datadog's CTO and co-founder, called *Monitoring 101* (https://www.datadoghq.com/blog/monitoring-101-collecting-data).

In this series, he recommends grouping observability data into three main categories: **work metrics**, **resource metrics**, and **system events**. In this section, we will dig into resource metrics and cover the USE method and how it applies to the autonomous service patterns and serverless. Specifically, we will look at capacity and concurrency limits, throttling and iterator age, and errors and fault events.

The USE method

Resource metrics describe the internal workings of our autonomous services. They tell us, at a lower level, how our services are using resources to produce their output. In traditional systems, these resources include CPU, memory, I/O, and so forth. For our serverless systems, our resources include the cloud services we use, such as datastores, streams, queues, functions, API gateways, and so on. Although these cloud services are fully managed, we still need to monitor and tune how we use them. This is where the USE method comes into play.

The **USE** method (https://www.brendangregg.com/usemethod.html) was developed by Brendan Gregg, based on his experience with managing systems at Netflix. This methodology recommends monitoring the *utilization*, *saturation*, and *errors* of our resources:

- **Utilization** – The percentage of capacity that is in use
- **Saturation** – When we exceed the capacity of a resource
- **Errors** – The count or rate of errors produced by the resource

Let's see how we can apply the USE method to our serverless resources.

Capacity and concurrency limits

Scalability is one of the main advantages of serverless resources. They have auto-scaling built in so that we do not have to worry about it. However, this scalability is not limitless. These resources usually have soft limits that are in place to protect us from runaway workloads that can drive up costs. These limits are set per account and region. This is one reason why we run our autonomous subsystems in separate accounts and across regions. This way, each subsystem has its own capacity, and they are not competing with each other for scarce resources.

Usually, the default limits provide enough capacity to support a single subsystem. However, increasing these limits is a time-consuming process that involves sending a support request to the cloud provider. Therefore, we need to monitor the **utilization** of these capacity limits, so that we have enough of an early warning to submit a timely support request. For example, AWS Lambda has a default limit of 1,000 concurrent function invocations. We should set up a threshold alert that lets us know when

utilization reaches 60% so that we can make the necessary adjustments to ensure we have plenty of headroom for traffic spikes.

We can also reserve capacity so that each resource, such as a function, has an allocated slice of the available capacity. However, this can be tricky because we can overallocate capacity and have it go underutilized and starve other resources of needed capacity. This is one advantage of using streams, such as AWS Kinesis, with our stream processors, because concurrency is automatically adjusted based on the number of stream shards.

We also collect capacity utilization resource metrics so that we can tune our services over time as we covered in the *Leveraging FinOps* section. We can use the historical capacity metrics to identify patterns and make informed decisions about worthwhile enhancements. For example, we may determine that we are overusing the capacity of a datastore like DynamoDB because we are using a feature inefficiently. In this case, we can make changes to improve performance and reduce cost.

Ultimately, we want to maintain an optimal amount of headroom in our capacity so that we avoid *saturation*, so let's see what happens when we miss the mark.

Throttling, iterator age, and queue depth

Saturation occurs when we exceed the capacity of a resource, and its ability to service requests degrades. **Throttling** errors and metrics are the most common and most obvious indicators that we are saturating a resource

with requests. In other words, we have exceeded 100% utilization of the allocated capacity.

Throttling may or may not be a problem. It may be temporary due to an unusual spike in traffic, or it may occur while a resource is auto-scaling or using its burst capacity to adjust to the normal ebb and tide of the traffic pattern. During these transient scenarios, we simply need to retry the requests using an exponential backoff to ensure that the requests go through. This capability is usually built into the cloud provider's SDK.

However, throttling may be an indicator of a more permanent condition. We may have already requested an increase in capacity since we are actively monitoring utilization. Or it may be an indicator of a bottleneck in our code that we need to fix. For example, a *command* function may have write-contention or a *query* function may have an inefficient query, which increases latency and the potential for concurrency. In either case, we may need to tune the data model.

Iterator age and **queue depth** metrics are other less obvious signs of *saturation* in our stream processors (that is, *listener* and *trigger* functions). These metrics tell us how long an event is waiting in a channel for a specific stream processor. High values indicate a potential bottleneck in these functions. We may not be processing events fast enough to keep up with the volume. We may be falling behind and may not be able to catch up unless we tune the throughput.

We covered various techniques for tuning stream processors in Chapter 4, *Trusting Facts and Eventual Consistency*. We may need to add rate limiting to reduce time wasted on retries or increase parallel requests to eliminate idle time. We could increase the batch size and the timeout so we can process more events per invocation. We may have too many consumers

competing for the resources of a single channel and need to split it into multiple channels. Or we may have more volume than expected and we need to add shards to a channel to increase the number of parallel stream processors. There are many dials we can turn. In general, we use these metrics to help us tune the system.

However, a high iterator age is not always the result of a performance problem. It can also be the result of errors and poison events. So, let's look at monitoring errors and fault events.

Errors and fault events

Errors can occur for many reasons and any resource in the call stack can produce errors. We monitor for errors so that we can more easily investigate them and determine their root cause. We group errors into two main categories: *retriable* and *non-retriable*.

Non-retriable errors are *permanent*. They need our attention before transactions can proceed. Bugs in our software are the primary cause of non-retriable errors. These are exactly the kinds of errors teams are worried about when they deploy enhancements.

We monitor for errors so that we can fix the bugs and *fail forward fast*. These are usually edge scenarios that we either missed, or we didn't think were likely.

We introduced **fault events** in *Chapter 4, Trusting Facts and Eventual Consistency*, to handle these kinds of errors in our stream processors (that is, listener and trigger functions). Bugs can create *poison events* that block good events. So, we set poison events aside by raising fault events so that good events can continue to flow. On the other hand, if an iterator age or

queue depth metric is unusually high and the stream processor is throwing errors, then it is an indicator that the function is not properly handling an unexpected error and raising a fault. So, we will monitor both scenarios.

Retriable errors are *transient*. For example, we just covered threshold errors in the previous section. We will automatically retry these calls, so that the system is *failure tolerant*. Our ESG services will experience retriable errors when an external system is having trouble. This is where the **Circuit-Breaker** pattern is useful, as we covered in *Chapter 7, Bridging Intersystem Gaps*. Our micro applications will experience retriable errors during a regional disruption, as we covered in *Chapter 9, Running in Multiple Regions*. However, we don't want to be satisfied with just retrying. We also track these retriable errors because there may be an underlying cause that we can address to continuously improve the system.

Now that we are collecting and organizing our metrics, let's look at how we can monitor and track them.

Tracking system events

Collecting and organizing our metrics is just the first step. Now we need to start using this data to see what we can learn about our services and how they perform in the real world.

We naturally find ourselves creating dashboards to visualize this data. We should have a high-level dashboard that gives us the status of the subsystem at a glance and a low-level dashboard that allows us to drill into the details. We should be able to filter these dashboards by the various tags, such as account, region, stage, service, function, and so forth. These dashboards will be invaluable when we need to do root cause analysis.

But we cannot keep our eyes on these dashboards all the time. We need the system to *watch* the metrics for us, *record* anything interesting, *send* us early warnings, and *page* us when it matters. In other words, to *fail forward fast*, we need to **monitor** our subsystem's resource metrics for various conditions. Here are several examples of typical monitors:

- Resource capacity utilization reaches 30% and 60%
- Iterator age exceeds 10 minutes
- Error rate exceeds 10% of requests
- Fault event count > 0
- BFF request latency 95 percentile > 1 second

You can find dashboard and monitor examples here: https://github.com/jgilbert01/templates/tree/master/template-datadog-resources.

When a resource metric meets a given condition then the monitoring system will *record* this as a system event. A **system event** represents a discrete point in time when something happened within the operation of the subsystem. On our dashboards, we can overlay the system events with the metrics to help us see correlations and patterns. We can also treat system events as *higher-order metrics* and look for patterns and trends in the system events. They help tell the story of what is happening in the subsystem.

However, we do not *page* the team on every system event. A spike in a resource metric may be temporary, auto-scaling may kick in and/or the resource may recover on its own. However we still record all system events because these additional data points can be helpful later to identify trends and to perform root cause analysis.

Yet, a system event may be an early warning sign of an impending problem, such as when utilization exceeds 30%. We may email the team and/or update the team's collaboration tool so that they are aware of the potential problem. We can automatically create a ticket in the team's tracking system so that they can prioritize and track the necessary tasks. We will also use system events to track when teams deploy changes since any change can have an unintended consequence and the deployment event can provide context for the root cause analysis.

We still need to tell teams when it is time to jump into action. Now that we are turning all this data into information, let's look at how we can make it actionable.

Alerting on work metrics

We set out to turn all our observability data into actionable information so that teams can experiment with the confidence that they can fail forward fast. But we need to **avoid alert fatigue**. If we over-alert, then the alerts become noise and the team will start to ignore them. As a result, their confidence will go down, their lead times will go up, and innovation will stagnate.

Instead, we need to zero in on the high-value metrics and only wake the team up in the middle of the night when these **Key Performance Indicators** (**KPIs**) go sideways. However, it is hard to determine which of the many metrics are the key metrics. So, let's start with some examples.

Netflix provides the classic example (https://netflixtechblog.com/sps-the-pulse-of-netflix-streaming-ae4db0e05f8a). They have identified one

metric as the most important indicator of a significant problem with their system. This one metric is the rate at which users press the *Play* button. They regionalize this metric and perform double exponential smoothing to find the range of acceptable behavior. They will *page* people in the middle of the night when they detect an anomaly in this metric. Other metrics may be important, but not worth waking people up for.

In our example Food Delivery system, the rate of the *OrderPlaced* and *DeliveryCompleted* domain events are likely to be important. These are examples of **work metrics**. Anomalies in these work metrics can indicate that there is a problem in their respective subsystems. These anomalies represent the **symptoms** of the problem, whereas *resource metrics* represent the root cause of the problem.

The general guideline is to page the team judiciously on symptoms rather than causes. All teams need to discover their most important work metrics so that they can page the teams about things that matter. In this section, we will dig into work metrics and cover the RED method and how it applies to our autonomous service patterns. Specifically, we will look at BFF service metrics, domain event metrics, and anomaly detection.

The RED method

Work metrics represent the useful output of our autonomous services. They tell us what our services are doing from the end-user's perspective. For example, are the users experiencing a responsive system or errors and increased latency? These metrics are the KPIs we will use to decide when we need to jump into action and fail forward fast. We will alert on work metrics for our BFF services because they are end-user facing, and for our

domain events since they represent the overall workflow and output of the system. This is where the RED method comes into play.

The **RED** method (https://grafana.com/files/grafanacon_eu_2018/Tom_Wilkie_GrafanaCon_EU_2018.pdf) was developed by Tom Wilkie, based on his experience at Weaveworks and Google. This methodology recommends monitoring *rate*, *error*, and *duration* metrics for each service:

- **Rate** – The number of successful requests/events per unit of time
- **Errors** – The rate of failed requests/events
- **Duration** – The amount of time (that is, latency) needed to process requests/events

Let's see how we can apply this method to our autonomous services.

BFF service metrics

In *Chapter 6, A Best Friend for the Frontend*, we covered the BFF pattern. The metrics of our BFF services are work metrics because these services operate at the boundary of the system. Their metrics are critical because the health of these services directly impacts the end users. When a BFF is unhealthy its metrics represent the symptoms of the underlying problem.

These services provide the queries and commands that the end users invoke through the frontend applications to perform the work they need. Each BFF service has an API gateway that these requests must pass through. We can use the API gateway's metrics as the work metrics for a given BFF service. It is important to note that the API gateway is also a resource that an autonomous service utilizes, but here we are viewing its metrics from a different angle.

The *rate* at which users invoke these queries and commands is a strong indicator of the importance of a service. Many services get invoked infrequently, while it is usually just a select few that get invoked continuously. Those few are usually the best candidates for alerting, mostly because of their importance. But the regularity of their usage patterns is also ripe for *anomaly detection*, as we will cover shortly.

The *error* rate of BFF calls is an important measure of the quality of the user experience. We won't directly alert the team based on error metrics, because the rate of successful requests will drop as errors increase. But they will certainly help lead us to the root cause. For example, 4xx errors will typically indicate that we have a bug in the front-end application that is making the calls. 5xx errors usually mean that there is a problem with the resources used by the service. We use these 5xx errors in our regional health checks as we covered in *Chapter 9, Running in Multiple Regions*.

The *duration* (that is, latency) of these calls is also an important measure of the quality of the user experience. Users do not like slow systems, so these metrics are a good place to look for opportunities for continuous improvement. Again, we won't alert the team on these metrics, but a sudden increase in duration will likely impact the rate of requests. Queries are the most likely requests to experience fluctuations in duration based on their complexity, but high-volume commands can block when there is resource contention, such as a hot shared. Commands also result in domain events, so let's look at domain event metrics next.

Domain event metrics

In *Chapter 4, Trusting Facts and Eventual Consistency*, we introduced stream processors and specifically listener functions, which consume

domain events, and in *Chapter 5, Turning the Cloud into the Database*, we introduced trigger functions, which produce domain events. The metrics of our stream processors are crucial indicators of the health of the system.

Our domain events are work metrics. They represent the output of the system. They state the facts about the work the system has completed. We can perform valuable business analytics based on domain events. We can also leverage domain events to monitor the health of the system.

The *rate* at which services emit various domain event types is a strong indicator of the importance of the work. Many domain event types are published infrequently, while it is usually a select few that are published continuously. Those few are usually the best candidates for alerting. Note that there is a strong correlation between BFF commands and domain events, so we can use domain events as a surrogate to avoid over-alerting.

The *error* rate and *duration* of stream processors is an important measure of the overall throughput of the system. We have already looked at fault events and iterator age in the *Collecting resource metrics* section.

Before we can use domain events as metrics, we need to send them to the monitoring system. To do this, we create a service with a function that consumes all events and writes out a count metric for each event. Exactly how we write out the metrics depends on the specific monitoring system. But in general, we want to create a count metric named **domain.event.count** with a standard set of tags. It is also helpful to create a similar metric named **domain.event.size** with the same tags so that we can gain an understanding of the size of our event payloads.

The most important tag is the **type** tag, which contains the value from the **event.type** field. The other tags help us understand the origins of the events,

such as **account**, **region**, **stage**, **source**, **functionname**, and **pipeline**. These values are available in our standard event envelope under the **event.tags** field.

You can find a template for collecting event metrics here: https://github.com/jgilbert01/templates/tree/master/template-event-lake-es.

Now, let's use our work metrics to alert the team.

Anomaly detection

Anomalies in our **work metrics** represent the symptoms of a problem with the system. These anomalies are an indicator that the team may need to act. In the Netflix example, they use double exponential smoothing to calculate the range of expected values. Fortunately, modern tools make it easy for us to accomplish the same result with less elbow grease. We can set up anomaly detection services, such as those provided by AWS CloudWatch or the Datadog monitoring service, to find the variations in our work metrics.

Anomaly detection continuously applies statistical and **machine learning** (**ML**) algorithms to find spikes and dips that deviate from the norm. When the service detects an anomaly in a work metric, it is time to alert the team so we can jump into action and address the issue before it seriously impacts end users. We will use the resource metrics and system events to help us quickly find the root cause of the problem.

Without these metrics, the users will most likely find an issue first, and it will take the team longer to resolve the problem. Over time, we will home in on the main work metrics that act as the KPIs, so that the team can move faster and faster.

Now, let’s look at observability for the frontend.

Observing real user activity

Up to this point, we have focused on the observability of our serverless backends. However, a significant portion of today’s application logic executes on the frontend within a user’s browser on their own devices. Leveraging the processing power of the user’s device allows us to spread out the load and dramatically improve the scalability of a system. However, it also makes it more difficult to monitor the behavior of the system, since we must capture information from many devices. To address this problem, we need **Real User Monitoring** (**RUM**) and synthetic transactions.

Real User Monitoring (RUM)

RUM encompasses the set of tools we use to observe the performance of the frontend application logic. This includes sampling page load performance metrics, capturing errors and console logs, and sending this information to the central monitoring system. Then we correlate the frontend metrics with the backend metrics to create a full picture of the performance of the system across the entire technology stack.

Adding RUM to a web application is straightforward. The following code block provides an example of initializing a third-party RUM tool:

```
import {datadogRum} from "@datadog/browser-rum";
datadogRum.init({
  applicationId: process.env.RUM_APP_ID,
  clientToken: process.env.RUM_CLIENT_ID,
});
```

This script simply loads and initializes the library with the necessary configuration settings. We can put this code in its own micro application and load it from the main micro application, that we covered in *Chapter 3*, *Taming the Presentation Tier*. As a result, all the micro applications will benefit from RUM.

RUM provides more than just technical metrics. It also provides functional metrics, such as page view counts and the time spent on a specific page. This information can help evaluate the outcome of experiments. For example, it can help us answer questions such as whether more people are using the new page versus the old page.

But RUM only provides information when we have real users exercising the system. We also need a control group of users that continuously exercises the system. We will cover this set of synthetic users next.

Synthetics

In addition to monitoring real users, we also want to simulate a set of users. We can think of this group of users as a scientific control group that allows us to set a baseline for the expected performance. To establish this baseline, we automate a set of synthetic transactions and execute them continuously, such as every 5 to 30 minutes. These transactions and the category of tools we use to create them are referred to as **synthetics**.

These synthetic transactions execute in a real browser, so they will generate the RUM metrics that we will use as the baseline. We also execute these transactions from multiple global locations to introduce different latency and network conditions. They also provide metrics during periods when there are little to no real user transactions.

Synthetics are essentially smoke tests. They allow us to continuously assess the stability of the system. But these scripts can be time-consuming to write and maintain, so we need to craft a strategic set of tests that provide the most bang for our buck.

At a bare minimum, we should implement an uptime check. This test simply loads the home page, checks for errors, and generates page load metrics. Going a bit further, we can navigate through the public pages. We should also test the sign-up and sign-in flows and then navigate through the protected pages. From here, we should also check the most essential transactions that the user cannot do without.

Now, let's look at how we can use our observability metrics for continuous performance tuning.

Tuning continuously

Continuous performance tuning is a natural benefit of the implicit scalability and high observability of serverless solutions because we can safely deploy with reasonable defaults and tune as we learn from the metrics. The resource and work metrics show us how our services actually perform, so we can make informed decisions about optimizing performance and throughput. For example, in *Chapter 4*, *Trusting Facts and Eventual Consistency*, we covered various techniques and parameters for optimizing the throughput of our stream processors. We will set these parameters upfront based on educated guesses and reasonable defaults. We can perform load testing to double-check these hypotheses. But it is ultimately the real observability metrics that dictate proper tuning.

Of all the autonomous service patterns, performance is of the utmost importance for BFF services, because they are user-facing, and users have high expectations. The work metrics are the place to start. If the user experience is sufficient, then it may not be worth spending the time and effort to tune the performance of a specific service. However, for serverless systems, tuning can have a significant positive impact on cost. So, we can experiment with optimizations until the cost benefits are no longer worth the effort.

We have already optimized the BFF service pattern for high performance by employing the CQRS pattern, which we covered in *Chapter 5, Turning the Cloud into the Database*. This eliminates competition for resources and allows each service to take full control of its resources. Latency-based routing helps to ensure that all users experience similar performance, as we covered in *Chapter 9, Running in Multiple Regions*. There is still more we can tune to improve performance. Let's look at some common optimizations to get you started. We can tune function memory allocation, minimize cold starts, avoid timeouts and retries, and leverage cache control.

Function memory allocation

The cost of running a function is typically calculated in gigabyte-seconds. The price of a gigabyte-second is based on the amount of memory allocated. The more memory allocated, the higher the price per gigabyte-second. However, the correlation between price and cost is counter-intuitive because memory allocation also corresponds to CPU power and network I/O throughput. This means that paying more per gigabyte-second may result in a lower overall cost and better performance.

For example, on AWS Lambda's pricing scale, if you double the memory allocation but consequently cut the execution time in half, the cost is the same. The corollary is that if you cut memory allocation in half and consequently double the execution time, then you are spending the same amount of money for less performance.

Start with a moderate allocation of memory as the reasonable default for all functions, such as 1,024 MB. Then focus on tuning the most expensive functions based on the production runtime metrics.

Cold starts

Cold start time is a common concern regarding the performance of **Function-as-a-Service** (**FaaS**). A cold start occurs when the FaaS service receives a request to execute a function and needs to initialize a new function instance first. This happens when there is a long period between invocations or when there are many concurrent invocations. The cold start time is caused by the additional steps that must be performed during initialization.

The first thing a new function instance must do is download the function bundle. The size of the bundle has an impact on the cold start time. Increasing the memory allocation of the function and thus the network I/O performance is the simplest way to increase the download speed. We also strive to minimize the size of the bundle. For example, all the templates provided with this book use the webpack module bundler, which performs tree shaking to remove unused code from the bundle.

Once the download is completed, the function instance can start the runtime environment. The choice of language runtime has an impact on the cold

start time. Scripting languages, such as JavaScript, are the best option because they do very little at startup. It is also important to avoid heavy-weight frameworks, such as dependency injection and object-relational mapping, which perform introspection on startup. The lightweight alternative we covered in the *Models and connectors* section of *Chapter 6, A Best Friend for the Frontend*, is more than sufficient for autonomous services.

Timeouts and retries

Networks are unreliable. For example, a call from a query function to the datastore can hang for no particular reason and the next call will go straight through. This scenario is common enough that libraries, such as the `aws-sdk`, have retry logic built in and turned on by default. However, we need to take control of the timeout period so that the end user does not need to wait long for the retry.

The following example sets the timeout for calls to DynamoDB:

```
class Connector {
  constructor() {
    this.db = DynamoDBDocumentClient.from(
      new DynamoDBClient({
         requestHandler: new NodeHttpHandler({
           requestTimeout: 1000,
           connectionTimeout: 1000,
    })}));
  } ...
}
```

The default timeout period in Node.js is 2 minutes. This is obviously much longer than a user is willing to wait. So, we need to set the `timeout` to a

more reasonable value. The value needs to be long enough to support the required queries and commands, but short enough to retry quickly.

Next, we need to set the timeout for the query and command function itself, so that it is long enough to support multiple retries. For example, the default timeout for an AWS Lambda function is 3 seconds. This provides the connector with enough time to perform at least one retry but may not be long enough for two retries.

Finally, the network performance of a function is associated with memory allocation. Lower network I/O capacity will increase the likelihood of failures and the need for retries. If a function is performing a lot of retries, then this is an indication that it needs more resources.

Cache-control

The response times of BFF service queries are highly optimized. A BFF service uses a high-performance cloud-native database, which it does not share, and we specifically design the materialized views to support the needed access patterns. This provides great performance for most usage scenarios.

However, for high-volume usage scenarios, we can further improve performance and control cost by adding `cache-control` headers and leveraging a **Content Delivery Network** (**CDN**). The following example sets the `cache-control` header on a query response:

```
api.get('/things/:id', (request, response) =>
  request.namespace.models.thing.get(request.params.id)
  .then((data) => response.status(200)
```

```
      .header('cache-control', 'max-age=3')
      .json(data)));
```

This example sets the cache value to a meager 3 seconds. However, for a high-traffic resource, this has the potential to significantly reduce the cost if at least one request has a cache hit. Adding `cache-control` headers to our responses tells a browser not to repeat a request and tells the CDN to reuse a response for other users. As a result, we reduce the load on the API gateway and the function and reduce the necessary capacity for the database, which reduces the cost for all these resources.

We covered the configuration of the CDN in *Chapter 9*, *Running in Multiple Regions*.

Summary

In this chapter, you learned why we need observability to fail forward fast and how serverless makes it easier to collect useful metrics. You also learned how to leverage FinOps and cost metrics to help decide when it is worth tuning our services.

We dug into the details, and you learned how to use resources metrics and the USE method, and work metrics and the RED method, to turn observability data into actionable information. You learned how to monitor the frontend with real user monitoring and synthetics. You also learned about continuous performance tuning and reviewed some common tuning tips.

In the next and final chapter, we will round out our discussion on architecting for innovation, with thoughts on getting started, gaining trust,

and building momentum. We will address some of the myths and anti-patterns that keep teams from getting off the ground and cover the various hurdles and impediments that teams must confront.

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.

**Limited Offer*

13

Don’t Delay, Start Experimenting

In *Chapter 12*, *Optimizing Observability*, we learned that our serverless resources emit many useful metrics that we can use to monitor our systems. We learned how to turn all this data into actionable information, by categorizing it as resource or work metrics so that teams have the confidence to experiment and fail forward fast.

Now, it is time for the last chapter. It is time to take everything you have learned and put micro frontends and autonomous service patterns into action. But where do we start? How do we build momentum? What pitfalls lie ahead? How do we cross the cultural divide between the old and the new? This chapter will answer these questions. But there will always be open questions. We can’t let them delay us. We need to start experimenting.

In this chapter, we’re going to cover the following main topics:

- Gaining trust and changing culture
- Funding products, not projects
- Dissecting the Strangler pattern
- Addressing event-first concerns
- Poly everything

Gaining trust and changing culture

Any new effort that goes against the grain of the current culture will experience a bumpy start. It's normal. But it won't take long for the culture to start changing once the new approach proves its worth.

I remember my first *lift-and-shift* project. It was a long uphill climb, but then things changed, almost overnight.

We initially transplanted our manual data center processes into the cloud. But we saw no real benefit from the cloud at that point because each release of the software was still slow and painful. So we set out to automate the deployment of the entire platform. We encountered resistance from the start. But we applied our agile, incremental development methodology and started to move forward. First, we did a quick proof of concept, and then we addressed incremental pain points as we progressed toward the full solution. Along the way, we picked up support.

After a long six months, we had all the support we needed, and we were ready to go into production. The initial deployment was painful, but it was successful. Everyone celebrated. Announcements and congratulatory emails went out. Then, we did another deployment on the next day, and more emails went out. And the same thing happened on the day after that. But on the fourth day, we did another deployment, and no emails went out. There was no big announcement, no congratulations, and no high-fives.

The culture changed that rapidly. It quickly became obvious that this was the new normal. We proved that we could perform deployments on any given day. A deployment was no longer a big event. It became routine. It made the whole process better for everyone. So everyone was on board.

This is a repeatable process. I have seen it happen on many products. We need to establish a vision, build momentum, and put an architectural runway in place. Let's look at these in more detail.

Establishing a vision

First and foremost, we need to establish an architectural vision for a system. It doesn't have to be perfect, but it does have to point the way forward. Without a vision, the system will easily devolve into a big ball of mud.

TIP

Have a look at these articles for more information: https://en.wikipedia.org/wiki/Big_ball_of_mud and http://laputan.org/mud.

In this book, I put forward the vision of the technical architecture of autonomous services and micro-apps, and *Chapter 11*, *Choreographing Deployment and Delivery*, provides a vision for the development methodology. But what we really need to move forward is a vision for the functional architecture of the system.

We need at least an initial sketch of the system divided into **autonomous subsystems**. We covered this topic in *Chapter 2*, *Defining Boundaries and Letting Go*, and we employed **event storming** to help us identify the core *actors* and the major *domain events*. With this information, we can begin to add structure to the system and visualize how the various pieces fit together.

From here, we can begin to craft a plan of action. The subsystem architectural vision leads to organizational structure and funding models. We will use it to organize our cloud accounts. The shared vision facilitates inter-team communication, and most importantly, at this point, it helps us identify a good place to start making progress. Let's see how we can start to build momentum.

Building momentum

We need to start small and build momentum. Don’t delay. Assemble the first team by seeding it with early adopters. These are skilled engineers who are excited about the possibilities. They will also be future leaders as we grow and add more teams.

We need to start with an experiment that immediately adds business value. For example, it could be a new feature that fills a troublesome gap in an existing business process. We cannot go a long time without delivering value to the business. Put a prototype of the frontend in the user’s hands as soon as possible, allowing them to see progress while you work on the backend behind the scenes.

This feature should be relatively independent so that we do not get bogged down. But it should be significant enough to demonstrate the value of the new approach. You can seed the feature using the templates provided in this book: https://github.com/jgilbert01/templates and https://github.com/jgilbert01/micro-frontend-template. It will be part of the first autonomous subsystem, so you can establish the process for stamping out the corresponding cloud accounts.

Find willing beta users. These users are eager to help improve the system because they see and feel the gaps in the existing system and want to make it better. They understand that this is a work in progress, and they want to be part of the experiment and vet the solution before you roll it out to a wider audience. They will also be a great source for more ideas.

Start building a roadmap of potential experiments. In *Chapter 11*, *Choreographing Deployment and Delivery*, we introduced *impact mapping* and *story mapping* as tools to identify valuable features and create a

roadmap. You will likely focus on one subsystem at first, but it is a good idea to put the walking skeleton for all the subsystems in place sooner rather than later.

And be prepared to get your hands dirty. We need to start putting an architectural runway in place for the teams and experiments that will come next so that they can get up and running smoothly. And the best way to know what works and what doesn't is to do it. Let's look at this runway next.

Constructing an architectural runway

In conjunction with the initial experiment of the previous section, we need to start putting an architectural runway in place. This runway consists of all the things that help the teams get up and running quickly and without duplication of effort. This consists of tools and templates, shared components, and more.

The initial experiment gives us an opportunity to put these capabilities in place. But we don't have much of a head start this first time around. So we need to work in parallel with the first experiment and try to stay one step ahead. You can use the templates provided with this book to get a head start. Each subsystem will need an event hub and a main entry point for its micro frontend. We will need the ability to stamp out cloud accounts for new subsystems and seed autonomous services and micro-apps from templates.

We can also start to put shared subsystems in place. For example, we will need a subsystem that encapsulates any interactions with the legacy systems, as we covered in *Chapter 7, Bridging Intersystem Gaps*. And we

will look at this more in the *Dissecting the Strangler pattern* section to come. We will also want a master data model subsystem that sits upstream and manages the reference data that all downstream services leverage. And all the way downstream, we will want subsystems for analytics and archiving. The initial team won't own these forever, but we need the skeleton in place for the initial experiment.

In *Chapter 2, Defining Boundaries and Letting Go*, we covered the various cross-cutting concerns that all teams will need in place, such as CI/CD pipelines and monitoring. You will want to put these in place quickly, so **Software-as-a-Service** (**SaaS**) solutions are the best way to go. You will also need to start looking at creating the teams that maintain your secured perimeter and your continuous auditing. Let's look at how we can build out teams next.

Seeding and splitting teams

Seed and split is an approach for knowledge sharing and cultural change. The team that works on the initial experiment will develop new muscle memory, gain valuable experience, and establish a new culture of continuous flow as they work through the various challenges. To field the next team, we split the first team in half and seed new members into both teams. The new members will benefit from the experience of the current members and learn the new culture.

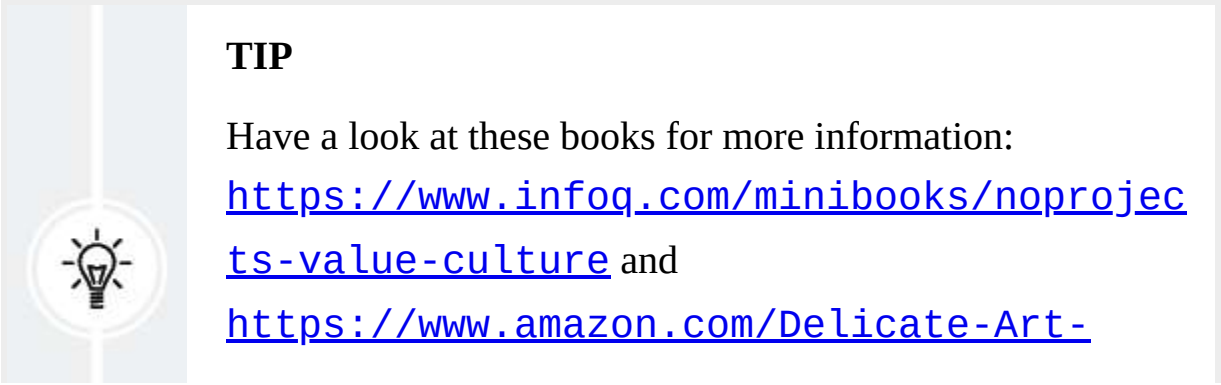
We can continue this process to build out the teams within each autonomous subsystem. Every new team should always start with several experienced members. To kick-start a new subsystem, several members can join a team in an established subsystem for a brief period and then bring the knowledge and culture back with them to start the new subsystem.

Within a subsystem, the upstream and downstream teams will work closely together, naturally share new experiences, and continuously improve the methodology and culture. The same thing will happen between autonomous subsystems as organizations work with upstream and downstream organizations to iron out overarching business processes. Slowly, but surely, the new culture will take hold as the teams continuously deliver business value and momentum builds.

We have established the vision for our architecture of autonomous subsystems, and we have kick-started our initial experiment with a team and architectural runway. Now, let's look at the funding model changes we need to ensure smooth sailing.

Funding products, not projects

In *Chapter 1*, *Architecting for Innovation*, we identified that traditional budgeting and planning practices increase lead times. These practices aim to control risk, but in reality, they increase the risk of delivering the wrong solution or delivering too late to be effective. The problem is that they focus on projects instead of products. They assume that we can define all the requirements upfront and provide realistic estimates. But we can't.



TIP

Have a look at these books for more information: https://www.infoq.com/minibooks/noprojects-value-culture and https://www.amazon.com/Delicate-Art-

> Bureaucracy-Transformation-Wrestler-ebook/dp/B086XM4WCK.

The bottom line is that we need to know how much money we are going to spend. Businesses need predictable cash flow. And we need to show that we are producing value and that the business is getting its money's worth. But it is hard to justify large multi-year budgets when we do not understand all the requirements upfront. We just can't.

Ultimately, business leaders need to know that everything is under control and that we can change course before we go off the rails and waste valuable time and effort. And this is exactly what our modern methods and architecture allow us to do. We are performing focused experiments so that we can learn and adapt. So what we need is a funding model that matches our development practices. We need a funding model that supports products instead of projects.

As architects, we need to take the lead and drive this change in approach. Business leaders know that a change is necessary, but they need guidance. The funding model I recommend is architecture- and team-driven. Let's look at these in turn.

Architecture-driven funding

In reality, a business is not going to stop spending money on software development. A business needs to deliver value to stay in business, and software is the foundation of modern business processes. But a business will reallocate funding away from failing projects and toward the best new project idea.

This is actually what we do when we experiment. We are allocating a little bit of money to an experiment and then reallocating as we learn. So our funding model needs to give teams the autonomy and authority to make these decisions. The business defines the objectives, and we determine the best way to meet those objectives through experimentation.

But the business needs to know what the money is being spent on, and this is where our architecture plays an all-important role. Our architecture defines what we are funding. It defines the autonomous subsystems and the high-level capabilities they provide. In other words, our architecture enumerates the business value it delivers. If it does not, then we need to go back to *Chapter 2*, *Defining Boundaries and Letting Go*, and revisit our architecture.

Our architecture defines the buckets of money that we are asking the business to provide so that we can experiment and continuously deliver business value. But now, we need to decide how much money to ask for. So let’s look at allocating capacity for autonomous teams next.

Team capacity-driven funding

Autonomous teams build software. An autonomous team is self-sufficient, full-stack, and cross-functional. In other words, a team has all the resources it needs to deliver value, and teams need team members. We, the members of a team, continuously deliver business value, and we can only do so much work in each period. So, a business needs to field enough teams to perform the work at hand.

This means that funding products ultimately comes down to deciding how much team capacity we want to allocate. On average, each team should

have the same capacity to produce value and require the same amount of funding. So once we determine how many teams we need, we can calculate how much funding we need.

We determine the number of teams we need, subsystem by subsystem. For example, is a subsystem big or small? Is its functionality stable, or is it evolving? There are many factors that can go into the number of teams we need to field for each subsystem. But doing this at the subsystem (that is, architecture) level helps control the scope and provides focus. We tally up the number of teams across the subsystems, and we have the total capacity that we need to fund.

Then, we let the teams turn the crank and perform their experiments. In *Chapter 2, Defining Boundaries and Letting Go*, we covered a continuous governance model that we can use to evaluate team performance. The teams determine how to allocate their capacity, and we use team metrics to help determine how much capacity to allocate going forward.

This approach gives the business what it needs: predictable cash flow and the confidence that the money is in good hands. And it gives teams what they need: clear objectives and autonomy so that they can drive down lead times and continuously deliver business value.

Now, let’s take a look at how we can migrate off our legacy systems.

Dissecting the Strangler pattern

The overarching theme of this book is that we want to move fast, but we do not want to break things. This applies to our legacy systems as well. Our

legacy architectures may impede change, but these systems provide valuable capabilities to our end users that they rely on every day. We must ensure that our legacy systems continue to work.

However, we do need to modernize our legacy systems, and they are filled with a lot of functionality. This means that if we follow a traditional approach, then it will take a long time to port the functionality, and in the meantime, the teams will not deliver any new business value. And when the time comes to cut over from the old system to the new system, this traditional all-or-nothing approach will put everything at risk. There is just too much risk to justify using the traditional approach.

Instead, we will build up new features and replacement features around the existing systems and integrate them. Slowly, but surely, we will chip away at the features of the legacy systems until we simply no longer need them. This is known as the **Strangler** pattern. It is a risk mitigation strategy that takes into account the fact that we need to continuously deliver business value without breaking things.

However, there is a right way and a wrong way to implement this pattern. Users prefer a seamless experience, and this creates the temptation to weave the new system into the old user experience. This usually involves writing an adapter layer that maps the old API to a new backend layer. And then we need to modify the legacy system so that it can toggle between the old and new endpoints.

This traditional approach does follow the **Substitution** principle, but it breaks the **Open-Closed** principle because we must modify the legacy systems. In addition, sooner or later, we will have to implement the new user experience and go through the risk of cutting over to that.

So we need an approach that does not require us to make changes to the legacy systems, just like we saw with the **Enterprise Application Integration** (**EAI**) approach that we covered in *Chapter 1*, *Architecting for Innovation*. We also need the ability to turn off the old systems **without any cutover**.

Moreover, the new features we create today will become tomorrow's legacy features. It is inevitable. So we want to put in place an approach that allows the system to naturally evolve. Fortunately, this is exactly what we designed our new architecture to achieve. So let's see how we can weave the old systems into the new system using our event-first and micro frontend architectures.

Event-first migration

Event-first migration is an integration-based approach for implementing the Strangler pattern. It is non-invasive to the legacy systems. In *Chapter 7*, *Bridging Intersystem Gaps*, we covered a variety of ways that we can leverage the **External Service Gateway** (**ESG**) pattern to create an anti-corruption layer around legacy systems.

We are essentially turning legacy systems into autonomous subsystems and services by wrapping them with a layer that produces and consumes events. It turns legacy data change events into external domain events for consumption by the new system. And it consumes external domain events from the new system and updates the legacy system. This creates a **bi-directional synchronization** between the new and old systems, using the **latching** mechanism that we covered in *Chapter 5*, *Turning the Cloud into the Database*.

This allows users to work in both the new system and the old systems. One user can start a business process in the new system and another user can complete the process in a legacy system, and vice versa. Beta users can experiment with the new version of a feature, and if it falls short, then they can complete their work in the legacy system, without losing a step.

Eventually, the legacy system will stop emitting events because we no longer use it. At this point, we can simply turn it off without any impact on the new system. Along the way, we delivered new business value without taking away valuable capabilities. We did not put the legacy system at risk, and we did not create technical debt in the new system. We controlled all the risks.

Still, some users will have to bounce back and forth between the old and the new user interfaces. If the legacy user interface uses web technology, then we can weave it into the new micro frontend.

Micro frontend – headless mode

During a legacy migration, we want the user experience to be as seamless as possible. In the early stages, some users may only use the new user interface, while beta users are happy to contribute and willing to bounce back and forth between the old and the new. As a migration progresses, more users will start to use the new interface, but they may use it in one setting and the old user interface in another setting. So it won’t feel like they are bouncing between the old and new.

As we roll the system out to more and more users, we want to provide them with one entry point to the system. We want this to be the new entry point. The old entry point will still be available but without all the new

capabilities. Our micro frontend approach, by virtue of its pluggability, is a natural fit. We just need to plug the old into the new.

We can accomplish this by writing thin micro-apps that wrap the legacy interface so that it snaps into the micro frontend. This is straightforward if the legacy frontend is also a single-page application. If the legacy interface uses server-side rendering, then it will involve a little more elbow grease, such as wrapping it in an iframe.

In either case, we need to make a slight modification to the legacy user interface to teach it how to run in **headless** mode. This is just a parameter that tells the application whether or not to render its banner and global navigations. The micro frontend will provide these instead when the legacy app executes as a micro-app. We will implement any other adaptations in the wrapper micro-app, such as transforming between different single sign-on approaches.

Again, sooner rather than later, the users will stop using these legacy screens, and we can simply unregister them with no impact on the new system. We did not force users to switch, and we did not take away the old interface. We gave users a reason to want to switch by providing new business value along the way. And we controlled all the risks.

But our legacy systems may not go completely away. They may enjoy a long retirement.

Retirement

Our legacy systems may still have a long life ahead. We will reduce our dependency on them and significantly reduce their runtime cost. They will no longer impede innovation. But they will probably still add value for

some time to come. This is good news for engineers and developers who are not ready to make the switch, as their skill sets will remain valuable.

There are two areas in particular where our legacy systems may still add value: *legacy interfaces* and *reporting*. Over the years, we likely developed many interfaces between the legacy systems and external systems. Replacing these interfaces will not happen overnight. It will take time to replace these, one by one. And external partners may not be willing to change on the same timetable. So the legacy systems can continue to provide these interfaces and simply act as a pass-through to the new system. In the meantime, we can de-provision all the unused resources to reduce cost.

The legacy database may also evolve into a reporting database. Likely, the legacy database already supports **Online Analytical Processing** (**OLAP**) and reporting. Now, it can continue to do so without competing for scarce database resources with the **Online Transaction Processing** (**OLTP**) features. It can sit downstream and support the analytics phase of the data life cycle, as we covered in *Chapter 5, Turning the Cloud into the Database*. Once we have ported the last interface and we have completely separated the OLAP and OLTP features, then it will be easier for the OLAP and reporting features to evolve and modernize.

The Strangler pattern allows us to control the risk of modernizing our legacy systems by incrementally porting the features until we can simply turn the legacy systems off. Now, let’s address some misconceptions about the event-first approach.

Addressing event-first concerns

Our event-first approach to building autonomous services is a bit of a paradigm shift. And as with any paradigm shift, there is plenty of opportunity for misunderstandings in the early adoption phases. This is completely understandable because we often have to unlearn the practices that we followed before. This can result in a resistance to change and the formulation of arguments against the change that turn out to be myths or anti-patterns.

As architects, we need to make the effort to understand the source of these concerns, so that we can set the record straight and lead the product in the right direction. To this end, I am including the most common misconceptions that I have encountered. Let’s start with a common misunderstanding of event lakes.

System of record versus source of truth

The **event lake** is a key element of our architecture. It provides a complete audit trail of all the actions and changes that occur within a subsystem. So it seems like a natural place for all services to pull their data from. However, that is not its purpose. We have optimized the event lake for writes as opposed to reads. We have optimized the event lake to ensure that we do not lose any data. We can go back to the event lake at any point in time and recreate the state of the system. This is an invaluable capability, but it does not support real-time access and it should not.

Instead, we have purposefully decomposed the system into a set of autonomous services along the data life cycle that act as various systems of record and sources of truth, as we covered in *Chapter 5, Turning the Cloud into the Database*. Upstream services in the *create* phase of the data life cycle act as the **system of record** for the data they produce and publish. The event lake collects and consolidates all this information into an ultimate record of truth, but we will not read from it. Instead, downstream services in the *use* phase of the data life cycle are responsible for collecting and caching the data they need to access in real time.

Further downstream, in the *analyze* and *archive* phases of the data life cycle, we design autonomous services that act as the **source of truth**, to support planning efforts, decision making, and records management. These services collect all the events and data from all the upstream services so that they can organize and transform all this data into valuable information.

All these autonomous services can evolve independently. The event lake provides the **system-wide transaction log**, which allows us to create an evolutionary system by turning the database inside out and turning the cloud into the database. It ensures that we can repair existing services and seed new services. But we don’t execute queries against the transaction log.

This naturally leads to concerns about duplicate data. So let’s address that next.

Duplicate data is good

We are purposefully replicating data across our autonomous services. This allows us to optimize each service and most importantly, it allows each to evolve independently. But this does lead to concerns about duplication of

data. This is a topic that we covered in detail in *Chapter 5, Turning the Cloud into the Database*, but it is important enough to repeat here briefly.

It turns out that there is a lot of duplicate data in our traditional relational database systems. There is the transaction log, which retains every action, and the tables, which retain the most recent state. Then there are all the indexes, plus the materialized views. Next, we need to replicate the data across three availability zones and multiple regions. Furthermore, to support more and more concurrent users, we need to create more and more read replicas. And when performance is still insufficient, we inevitably create a read-through cache for each application.

So, we really aren't creating more duplicate data. We are just making the replication more visible and taking full control of it. And we may actually be creating less duplication, since we keep the data in each autonomous service very lean. But even if we do create more duplication in some cases, it is worthwhile, as we are enabling autonomous teams so that they can drive down lead times and increase the pace of innovation.

There is also a similar concern that we are duplicating software. So let's address that as well.

Avoid false reuse

Software reuse is a double-edged sword. We are all taught as entry-level programmers to decompose our programs into reusable functions so that the code is easier to understand and maintain. In other words, reuse is supposed to help reduce lead times. But reuse also increases coupling. When multiple pieces of software share another piece of software, the shared software cannot change without careful consideration of the impact on all the dependents. This increases the need for inter-team coordination, which in

turn increases lead times. Therefore, we have to make sure that we reuse software wisely so that the liabilities of reuse do not outweigh the benefits.

As a general practice, we do not reuse software between autonomous services. Within the code base of a single service, we certainly strive for proper decomposition with **Model** and **Connector** classes, as we covered in *Chapter 6, A Best Friend for the Frontend*. But we defer commitment to creating and using shared libraries between services. We will refactor as we learn more about the functionality.

This helps us avoid the false reuse that ties systems into knots and hinders their evolution. **False reuse** occurs early in the development of a system before we have discovered the bulk of the requirements. We look across a system and we see what looks like duplication. But we need to resist the urge to consolidate. If we are patient, we will find that many of these similar features will diverge as the needs of their individual actors emerge.

In hindsight, false reuse is easy to spot. We can find it in those shared libraries with all the complex conditional logic that no one is willing to change. As the requirements rolled in, we had to go back to what we thought was common code and torture the logic with all the minutiae of the individual actors. It inevitably turned into a convoluted set of **if…then** statements that are brittle and impossible to understand.

Instead, we will wait, and we will learn. The boundaries of our autonomous subsystems and services make it difficult to fall into the trap because the code lives in separate repositories. When we do find real reuse, then we can refactor. We can create a shared library and retrofit the individual services and micro-apps when and if they need to change. We can move logic upstream and produce events so that downstream services can react. But

more likely, the different actors really do have different requirements, and the code that looks similar now is just boilerplate.

Now, let's look at whether or not we should standardize our tools and technology, or use what works best case by case.

Poly everything

We all have our preferences. You prefer a specific programming language, a specific database, and a specific cloud provider. You have lots of experience with them, so you can get the job done fast. Everyone should follow suit. Everyone should just get along and use the same tools and technologies. Or it should be the Wild West, and everyone should be an "outlaw" and do what they want to. Well, of course, neither of these extremes is practical. There are limits. There must be some governance. But as we covered in *Chapter 2, Defining Boundaries and Letting Go*, governance can impede innovation. So we want a balanced portfolio. We need diversification. We do not want to impede. And we need to be prepared for what might come. Let's see how we can strike the right balance with a poly-everything approach.

Polyglot programming

Polyglot programming is the idea that we want to use the right programming language for the job at hand. The different programming languages have their strengths and weaknesses, so we want to optimize the whole and use the different languages where they work best.

You may have noticed that the examples in this book are written in **JavaScript**. As it turns out, JavaScript is a great match for stream

processing, with its functional reactive programming style and asynchronous non-blocking I/O. Plus, **JavaScript Object Notation** (**JSON**) is native, so there is no need for additional transformations. And its lightweight startup characteristics are perfect for serverless.

You are likely thinking, *"But we are a Java shop. We don't have JavaScript developers."* But that is not exactly true. Most new user interfaces are written in JavaScript, and all those skills are transferrable to writing the serverless **Backend for Frontend** (**BFF**) services that we covered in *Chapter 6, A Best Friend for the Frontend*. An autonomous team of JavaScript developers can implement the full stack, from the micro-app to the BFF service and beyond. This is a very powerful incentive from a budgeting and capacity perspective.

Personally, I started using Java when it was born in the 1990s and thought I would never need another programming language. Then, SPA applications and serverless arrived on the scene, and I had to switch. But JavaScript had matured significantly, and I was amazed at how powerful and easy it was. I haven't looked back since. Most Java developers are interested in expanding their skills, and there are countless stories like mine. But as we just mentioned in the *Dissecting the Strangler pattern* section, our legacy systems aren't going away any time soon, so having both JavaScript and Java skills is valuable.

All that said, machine learning is becoming more and more important. And the tools around this capability tend to be written in **Python**. So, when working in this area, it does not make sense to go against the grain and force-fit JavaScript or Java into the mix. And who knows what is coming next. We don't want to be in a situation where we only have a limited set of skills.

Our autonomous services provide natural boundaries for defining the use of a programming language because they each have their own repository and build process. We can choose the right programming language on a service-by-service basis. If a service needs two programming languages, then we should probably split it into two services. We can write our BFF services in JavaScript, and we can let the requirements of our ESG and control services dictate the best choice. And this is all possible because our domain events in between the services are completely agnostic to these decisions.

Now, let's see how this approach applies to the database layer as well.

Polyglot persistence

Polyglot persistence is the idea that we want to use the right database technology for the work at hand. In *Chapter 5, Turning the Cloud into the Database,* we identified the fact that over the data life cycle, different actors use the same logical data in different ways, with different access patterns and requirements. This means that different database technologies may be better suited for the different data life cycle phases.

Each autonomous service maintains its own data. This allows us to choose the right database technology on a service-by-service basis. Fortunately, our *serverless-first* approach facilitates polyglot persistence, as we are offloading the heavy lifting of running these databases to the cloud provider. This allows the teams to focus on continuously delivering business value.

But vendor lock-in is a common concern, even though it doesn't need to be. We are making these decisions per service, which means we can change direction per service. This is not a monolithic decision that we must make

for the whole system. We are also structuring the code within each service to support making a switch.

We covered **Model** and **Connector** classes in *Chapter 6, A Best Friend for the Frontend*. The **Model** classes contain all the business logic and delegate to the connectors to make external calls. The **Connector** classes encapsulate our polyglot persistence decisions. For example, each cloud provider offers similar services. We can easily swap a connector for AWS DynamoDB with a connector for Azure Cosmos DB. This is the **Substitution** principle at work again.

This leads naturally to the decision of picking a cloud provider. Let's see how this approach scales up.

Polycloud

Polycloud is the idea that we want to use the right cloud provider for the work at hand. On a service-by-service basis, we choose a cloud provider to take advantage of a distinguishing capability only offered by that cloud provider. In practice, this means we pick a cloud provider at the autonomous subsystem level, even if it is a subsystem of only one service. We covered the approach for integrating between cloud providers in *Chapter 7, Bridging Intersystem Gaps*.

Polycloud is different from the notion of multi-cloud. Multi-cloud suggests that a service should be capable of running on multiple cloud providers simultaneously, supporting a scenario where a cloud provider fails across all its regions at once. But this scenario has never happened. And few services can justify the expense. At the very least, it suggests that we should be able

to lift and shift services from one cloud provider to another. But this is not practical or realistic either.

As it turns out, the code we write and run in the cloud is just the tip of a very big iceberg. Below the surface, there is a great deal of infrastructure that supports our services and applications. All the cloud providers offer these capabilities, but we should not underestimate their differences and the elbow grease that goes into provisioning this infrastructure. There are vendors that offer compatibility layers, but this is vendor lock-in to a smaller third party.

Instead, we want a balanced portfolio with diversification across cloud providers. For example, we could aim for an 80:15:5 ratio. We should have a preferred cloud provider for most of our services. Then, we build up competence across two other cloud providers by using them to run a subset of services. This allows us to gain knowledge and put the architecture in place to quickly move services between cloud providers. Our *serverless-first* approach offloads the bulk of the iceberg to the cloud provider, while our connector classes and handler functions encapsulate the cloud provider-specific interfaces. Our observability tool should give us a single pane of glass across cloud providers, and our CI/CD tools should run on a provider that is different from the services it deploys.

All in all, there is room for many tools and technologies. When choosing between the options, we should select the ones that help us drive down lead times and continuously deliver business value.

Summary

In this final chapter, we covered some final thoughts on making progress on the path of architecting for innovation. We covered the importance of gaining trust and building momentum, as well as constructing an architectural runway. We also learned how to mold our funding model into our process of continuous experimentation and discovery. We dug into the Strangler pattern and how it reduces migration risk, and we addressed common misconceptions about the event-first approach. We also learned how polyglot programming, polyglot persistence, and polycloud help us optimize a system as a whole and prepare for the future.

As a next step, give the various templates a try. Create an event hub and stand up a micro frontend. Then, create a micro-app and a BFF service. Wrap an external system with an ESG service, and have it react to the events from the BFF service, and vice versa. Along the way, start thinking about your system and how you would divide it into autonomous subsystems. Who are the major actors, and what domain events do they elicit? What are the core capabilities? How does this all fit together? Then, start thinking about a plan of action. Most of all, have fun on your journey.

Leave a review!

Enjoyed this book? Help readers like you by leaving an Amazon review. Scan the QR code below for a 20% discount code.

**Limited Offer*

packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

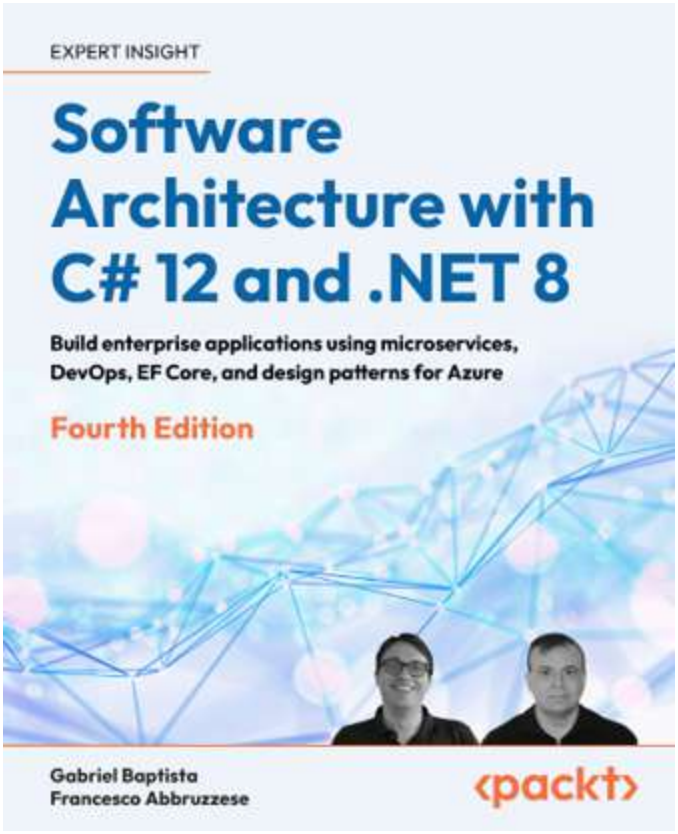
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

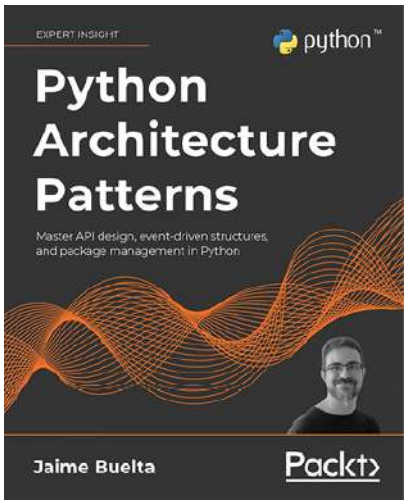


Software Architecture with C# 12 and .NET 8

Gabriel Baptista

Francesco Abbruzzese ISBN: 9781805127659

- Program and maintain Azure DevOps and explore GitHub Projects
- Apply architectural approaches such as layered architecture
- Make effective choices between cloud-based and data storage solutions
- Use tools such as containers to manage microservices effectively
- Understand when it is best to use test-driven development (TDD)
- Implement frontend microservices, worker microservices, and distributed transactions
- Get a handle on DevOps principles to enable CI/CD environments



Python Architecture Patterns

Jaime Buelta

ISBN: 9781801819992

- Think like an architect, analyzing software architecture patterns
- Explore API design, data storage, and data representation methods
- Investigate the nuances of common architectural structures
- Utilize and interoperate elements of patterns such as microservices
- Implement test-driven development to perform quality code testing
- Recognize chunks of code that can be restructured as packages
- Maintain backward compatibility and deploy iterative changes

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Software Architecture Patterns for Serverless Systems, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A abort event 272-274

ACID 2.0 275, 276

Action BFF service 211, 212

active/active-secondary 290

balanced data processing 291

balanced users 291

adapters 218

agile method 5, 360

Amazon Web Services (AWS) 21, 385

anomaly detection 409, 410

anti-corruption layer 30

creating 217, 218

external bulkhead 220

macro-level ports and adapters 218, 219

API Gateway 197

global endpoints 304, 305

regional endpoints 302, 304

API keys using 352

Application Programming Interface (API) 242

architectural boundaries defining 28

architectural runway constructing 420

architecture-driven funding 422

Archive BFF service 214, 215

artificial feature flag 363

asynchronous contract 389, 390

asynchronous inter-service communication 17

Asynchronous JavaScript and XML (Ajax) 68

auditing continuously 352

audit service 354

autonomous service CI/CD pipeline and GitOps 56

- dissecting 55
- functions 58
- persistence 57
- repository 55
- shared libraries 61
- stack 56
- tests 56
- trilateral API 58

autonomous service patterns 53

- Backend For Frontend (BFF) pattern 53
- Control Service pattern 54
- External Service Gateway (ESG) pattern 54

autonomous services asynchronous inter-service communication 17

bulkheads, creating 15, 16

data life cycle 22, 23

disposable architecture 21

event-driven enterprise 18

first-class citizen events 19

fortified boundaries 17, 18

idempotence and ordered tolerance 20

inversion of responsibility 19

knowledge, creating 20

micro frontends 23, 24

observability 24, 25

organic evolution 25

self-service 20

used, for enabling autonomous teams 14, 15

autonomous subsystem 329, 419

autonomous service patterns 53

context diagram 50, 51

dissecting 50

event hub 52

micro frontend 51

topology 230

autonomous teams enabling, with autonomous services 14, 15

AWS Certificate Manager (ACM) 337

aws-kms-ee library 348

aws-lambda-stream library 348

B

Backend for Frontend (BFF) pattern 16, 30, 34, 53, 254, 301, 367, 429

Backend for Frontend (BFF) service Action BFF service 211, 212

- Archive BFF service 214, 215
- CRUD BFF service 207
- Dashboard BFF service 212
- implementing 207
- JWT, passing to 342
- List-of-values (Lov)) BFF service 208, 209
- Reporting BFF service 213, 214
- Search BFF service 210
- securing 342
- Task BFF service 209

Backend for Frontend (BFF) service pattern 192

- API Gateway 197
- command function 198
- datastore 196
- dissecting 195, 196

listener function 198, 199
query function 198
trigger function 199
balanced data processing 291
balanced users 291
batching 144
batch integration 10
beta users 377
BFF service metrics 408
BFF services, security JWT assertion 343, 344
JWT authorizer 342, 343
JWT filter 344
last modified by 345
least privilege 346
bi-directional synchronization 424
boundary service pattern 191
bounded context 30
branch-by-abstraction approach 33
build-time auditing 352
bulkheads 397
business processes entry event 265-268
exit event 265-268
orchestrating 264, 265
parallel execution 269, 270

business process management (BPM) 265

business value delivering 2

high-velocity teamwork 2

C

cache-control headers 92

Cache-First strategy 90

callback function 227

CDN plus API Gateway 306-308

change data capture (CDC) 115, 152, 197, 236, 237

database-first event sourcing 183-186

latching 188, 190

mechanism 57

soft deletes 186-188

stream 258

using 182

change event replication versus domain event replication 310

chief technical officer (CTO) 2

choreography cons 254-256

pros 254-256

CI/CD pipelines 56, 378

deployment service permissions 333, 334

permission boundaries 334-336

permissions 332, 333

securing 332

Circuit-Breaker pattern 239, 240, 405

cloud accounts accounts-as-code 329, 330

securing 329

securing, with identity access management 330, 331

cloud database derived data 157, 158

transaction log 156, 157

turning 155, 156

cloud providers subsystem, integrating across 234, 235

cloud regional health checking 294

regional health checks 295

traditional health checks 294, 295

cold start 413

Command, Publish, Consume, Query (CPCQ) 6, 61, 115, 160, 293

command/query function 61

Command Query Responsibility Segregation (CQRS) pattern 115, 158, 159, 191

capacity per query 165

capacity per reader 165

inbound bulkheads 162, 163

live cache 164

materialized view 160, 162

system-wide CQRS 159, 160

complex event processing (CEP) logic decision tables 279, 280

implementing 279

missing events 281, 282

Connector classes 430

consistency 102

chain reaction 103

facts 102

transparency 102

Content Delivery Network (CDN) 211, 299-301, 336, 414

context diagram 51

context mapping 50

context maps 30

Continuous Deployment (CD) pipeline 391, 392

regional canary deployments 392, 393

synthetics 393, 394

continuous deployment test continuous discovery 358, 359

continuous testing 359, 360

optimizing 358

continuous discovery 358, 359

continuous integration and continuous deployment (CI/CD) 359

Continuous Integration (CI) 28

Continuous Integration (CI) pipeline 378, 379

contract testing 387, 388

end-to-end testing 390, 391

integration testing 385

serverless testing 380

serverless testing honeycomb 380-382

unit testing 383

continuous performance tuning 412

cache-control 414, 415

cold starts 413

function memory allocation 412, 413

timeouts and retries 413, 414

continuous testing 359, 360

contract testing 387, 388

asynchronous contract 389, 390

synchronous contract 388

Control Service pattern 16, 54, 254

collate 259

collect 258

correlate 258

dissecting 256, 257

emit 262

evaluating 260, 261

expire 263, 264

Conway's Law 45

costs on cloud invoice expected cost 401

unexpected or hidden cost 400, 401

coverage 383

Create, Read, Update, and Delete (CRUD) 207

Cross-Origin Resource Sharing (CORS) 306

cross-site scripting (XSS) 336

CRUD BFF service 207

crypto-shredding 349, 350

culture changing 417, 418

D

Dashboard BFF service 212

Database Migration Service (DMS) 237

data challenges cross-referencing 248, 249

data, enriching 247, 248

idempotence 246, 247

latching 248, 249

slow data resync 249, 250

tackling 246

data gravity 148

competing demands 148, 149

insufficient capacity 149

intractable volumes 150

data lake 116

data lean 165, 166

projections 166, 167
time to live (TTL) 167, 168
data life cycle (DLC) 150, 151
analyze phase 153, 154
archive phase 154, 155
create phase 151
use phase 152, 153
data modeling for operational performance 174
aggregates 174, 175
edges 174-176
nodes 174-176
partition key (pk) 176
sharding 176
single table design examples 177, 178
data store 18
Dead Letter Queues (DLQ) 319
decision tables 279, 280
dependencies 8, 9
Dependency Injection (DI) 34
dependency inversion principle (DIP) 34, 35, 266
deployment service 332
derived data 157, 158
design-by-contract approach 33
developer experience (DX) 17

direct SQL 238

Distributed Denial of Service (DDoS) attacks 336

domain aggregate 30

Domain-Driven Design (DDD) 29, 50

bounded context 30

domain aggregate 30

domain event 30

domain event replication versus change event replication 310

domain events 30, 34, 107

event carried state transfer 109

event envelope 107, 109

internal, versus external 111

metrics 408, 409

substitution 110

Domain Name System (DNS) 305, 336

domain-specific language (DSL) 261

downstream subsystem ingress flow 232, 234

DynamoDB Accelerator (DAX) 153

E

egress function 54

Elasticsearch 210

end-to-end testing 390, 391

Enterprise Application Integration (EAI) 11, 236

approach 424
enterprise service bus (ESB) 13
ESG services API keys, using 352
securing 350
shared secrets, securing 350, 351
event audit modifying 353
event bus 107
Event Catalog reference link 58
event-driven architecture (EDA) 18
event-first 38, 39
verbs, focusing on 40
event-first approach 424
concerns, addressing 426
duplicate data concerns 427
false reuse, avoiding 428
system of record, versus source of truth 426
event hub 18, 52
channel topology 111, 112
domain events 107
publishing to 105-107
routing 111
services, connecting through 43
event lake 116, 426
events, replaying 117, 118

indexing events 117

perpetual storage 116, 117

events 32, 33

responsibility for invocation, inverting 42

treating, as contracts 41

treating, as facts instead of ephemeral messages 40, 41

Event Sourcing pattern 191

dissecting 113, 114

event lake 116

systemwide event sourcing 114, 115

event-sourcing snapshots ACID 2.0 275, 276

calculating 274, 275

functional requirement 276

snapshot event 276, 278

events topic 95

event storming 39, 40, 419

event stream 118

concurrency control 121

connectors 130, 131

creating 127

filtering 129

mapping 129, 130

micro batching 122, 123

micro-event stores 121

multiplexing 129
processing 122
programming paradigm, selecting 124
stream-first event sourcing 119, 120
temporal storage 118
unit of work (uow) 128
eventually consistent business process 100
concurrency and partitions 103
consistency 102
parallelism 104, 105
staging 100
tolerance and idempotence 104
exploratory testing 377
external domain events 34, 50
external global endpoints 321, 322
External Service Gateway (ESG) pattern 16, 38, 50, 53, 54, 218, 220, 254, 320, 367, 424
connectivity 220
packaging 224
semantic transformation 221
separate cloud accounts 224, 225
Extract, Transform, and Load (ETL) 22, 238

F

fail forward fast 363

failure design 131

backpressure 132

fault events 134, 135

poison events 133

rate limiting 132

resubmission 136

false reuse 428

fault events 134, 135, 405

feature flag 362

artificial feature flag 363

keystone feature flag 362

natural feature flag 362

feature flipping 376

beta users 377

exploratory testing 377

general availability (GA) 377

FinOps cost, using as metric 399

monthly serverless cloud invoice, dissecting 400

using 399

worth-based development 401

first-in, first-out (FIFO) 104

first stream processor 172

forward fast failing 398

frontend monolith splitting 71

splitting, by device type 72

splitting, by subsystem 71

splitting, by user activity 71, 72

splitting, by version 73

frontend team 194

Functional Reactive Programming (FRP) 101, 124

Function-as-a-Service (FaaS) 18, 58, 122, 198, 413

function-level nano architecture connectors 201, 202

dissecting 200

handlers 202, 203

models 201

functions architecting, at micro level 60, 61

architecting, at nano level 59

G

Gatsby reference link 69

general availability (GA) 378

General Data Protection Regulation (GDPR) 349

GitOps 56

global bus 308, 309

global/edge infrastructure 336, 337

global endpoints 304, 305

governance costs 400

governing without impeding innovation 62

automation and cross-cutting concerns, providing 62, 63

culture of robustness, providing 63, 64

team metrics, harnessing 64, 65

graphical user interface (GUI) 11

GraphQL 58, 205-207, 245, 366

grouping 144

H

hardware provisioning 6

hexagonal architecture 35, 37

function-level (nano) 38

service-level (micro) 38

subsystem-level (macro) 38

HyperText Transfer Protocol (HTTP) 385

I idempotence and order tolerance deterministic identifiers 169

immutable event triggers 172-174

implementing 168

inverse optimistic locking 170-172

identity access management 330, 331

impact mapping 370

Imperative Programming 124

Infrastructure as Code (IaC) 6, 62

ingress flow 235

ingress function 54

Initiator BFF service 266

integration events 50

integration styles batch integration 10

dissecting 9

enterprise application integration (EAI) 11

microservices 13, 14

real-time integration 11

service-oriented architecture (SOA) 13

shared database 12

spaghetti integration 10, 11

integration testing 385

VCR test double 385-387

Interface Segregation Principle (ISP) 28, 33, 34

internal domain events 34, 49

inter-service collaboration promoting 254

intersystem differences addressing 320, 321

external global endpoints 321, 322

legacy regional endpoints 322

inter-team communication 8, 9

inverse-oplock technique 313

Inversion of Control (IoC) 34

inversion of responsibility principle (IRP) 35, 115, 258, 266

iterator age 404

J

Jamstack reference link 69

Jamstack application 72

Java Messaging Service (JMS) 240

JavaScript 429

JavaScript Object Notation (JSON) 429

JSON Schema reference link 58

JSON Web Key Set (JWKS) 343

JSON Web Token (JWT) 337

passing, to BFF services 342

JWT assertion 343, 344

JWT authorizer 342, 343

JWT filter 344

K

Kanban method 5

key performance indicators (KPIs) 24, 360, 406

keystone feature flag 362

L

lambda 59

latching 189, 248, 249

latching mechanism 424

latency-based routing 291

lead time decision making 4

dependencies 8, 9

dissecting 3

inter-team communication 8, 9

risk mitigation 4

software deployments 6

Software Development Life Cycle (SDLC) methodology 5

software structure 7

traditional testing method 8

lean materialized views 57

Lean method 5

legacy regional endpoints 322

legacy systems subsystems, integrating with 235, 236

Liskov Substitution Principle (LSP) 32, 268

listener function 53, 54, 61, 198, 199, 235

List-of-values (Lov)) BFF service 208

Little's Law 136

live updates, offline-first long polling 96

WebSocket 95

local cache, offline-first 88

code, caching 88-91

data reads, caching 91, 92
data writes, caching 92-94

M

machine learning (ML) 410
models 283
predictions 284
using, for control flow 283
main app 75, 76
importmap file 76, 77
index file 76
registration 77, 79
Markdown markup language 69
mean time to recovery (MTTR) 24, 360, 398
metrics 295-297
micro-app 79
activation 80
entry file 79, 80
life cycle 81
manual activation 82
root component 80
route-based activation 81
micro-event store 121
micro frontends 32, 51

benefits 74, 75
dissecting 73
inter-application communication 85, 86
main app 75, 76
Manifest deployer 85
micro-app 79
mount points 83-85
reference link 75
microservices 13, 14
missing events 281, 282
mock object 384
Model classes 430
momentum building 419
multi-factor authentication (MFA) 329, 331
multi-master replication 311-313
multiple levels experiments 369, 370
planning 368, 369
story backlog 370
task roadmap 372, 373
multi-regional cron jobs implementing 323
job records and events, inserting 324
replication and idempotence, relying 324, 325
multi-regional deployment higher user satisfaction 289, 290
justifying 288

nominal cost 289
regional disruption 288

N

natural feature flag 362
Network-First strategy 90
Next.js reference link 69
non-retriable errors 404

O

observability making easy 398, 399
offline-first designing for 86, 87
live updates 94
local cache 88
transparency 87
Online Analytical Processing (OLAP) 22, 151, 426
Online Transaction Processing (OLTP) 22, 151, 426
Open API egress query 245
egress webhook 244
ingress command 243, 244
ingress event 242
providing 242
reference link 58
Open-Closed Principle (OCP) 32, 423

OpenID Connect (OIDC) protocol 337-339

Open SPI providing 242

Open Worldwide Application Security Project (OWASP) 336

P

parallel execution 269, 270

partition key (pk) 176

perimeter encryption, in transit 337

federated identity 337

global/edge infrastructure 336, 337

securing 336

permission boundaries 334

persistence 57

Personal Identifiable Information (PII) 328

pipeline service 332

poison events 133

polycloud 234

poly-everything approach 429

polycloud 431

polyglot persistence 430

polyglot programming 429, 430

polyglot persistence 151

ports 218

Ports and Adapters 36

presentation tier conditional rendering 340, 341

JWT to BFF services, passing 342

OpenID Connect 338, 339

protected routes 341

securing 337

primary/hot-secondary 290

products funding 421

architecture-driven funding 422

team capacity-driven funding 422, 423

programming paradigm functional reactive programming 125

imperative programming 124

stream processing 125, 126

Progressive Web Applications (PWAs) reference link 70

protracted eventual consistency 293, 294

Python 430

Q

quality assurance (QA) 24

queue depth 404

R

real-time integration 11

real user activity observing 410

Real User Monitoring (RUM) 410, 411

synthetics 411

Real User Monitoring (RUM) 410, 411

receive function 243

RED method 407

regional canary deployments 288, 392, 393

regional endpoints 302, 304

CDN plus API Gateway 307, 308

global bus 308, 309

regional failover command failover 317, 318

dissecting 315

listener failure points 319, 320

offline-first 293

preparing 292

protracted eventual consistency 293, 294

query failover 315, 316

trigger failure points 318, 319

regional health checks metrics 295-297

service 297, 298

regional routing API Gateway 301

CDN plus API Gateway 306

configuring 299

regional topology active/active-secondary 290

primary/hot-secondary 290

selecting 290

regions replication 310

change event replication, versus domain event replication 310

multi-master replication 311-313

round-robin replication 313-315

relay 240, 241

Reporting BFF service 213, 214

repository 55

Representational State Transfer (REST) 203-205, 365

request function 244

resource metrics capacity and concurrency limits 403

collecting 402

errors 404

fault events 405

iterator age 404

queue depth 404

saturation 403

USE method 402

retriable errors 405

risk mitigation 360

fail forward fast 363

feature flag 362

release deployment, decoupling 361

small batch size 360

Robustness principle 63, 364, 365

root-cause analysis (RCA) 376

round-robin replication 313-315

runtime auditing 352

S

Saga pattern abort event 272-274

employing 271

transactions, compensating 271

saturation 403

Search BFF service 209, 211

seed and split approach 420

self-service cloud 9

semantic transformation 221

egress flow 223

ingress flow 222

sensitive data envelope encryption 347-349

redacting 346, 347

serverless 328

serverless-first approach 23

Serverless Framework (SLS) 333

serverless testing honeycomb 380-382

service-level agreement (SLA) 264

service-oriented architecture (SOA) 13

Service Provider Interface (SPI) 242

session consistency 92

sharding 176

shared database 12

shared responsibility model 328

shared secrets securing 350, 351

single-page application (SPA) 23, 68, 299

Single Responsibility Principle (SRP) 22, 31, 148, 193, 217

Single Sign-On (SSO) 331

Single-spa reference link 75

single table design examples 177, 178

- cart service 180, 181
- customer service 179, 180
- delivery service 181, 182
- restaurant service 178, 179

single user activity 193

snapshot event 276, 278

Software as a Service (SaaS) 117, 225, 226

software deployments 6

Software Development Life Cycle (SDLC) methodology 5

software structure 7

SOLID principles 30

- Dependency Inversion Principle (DIP) 34, 35
- Interface Segregation Principle (ISP) 33, 34
- Liskov Substitution Principle (LSP) 32

Open-Closed Principle (OCP) 32
Single Responsibility Principle (SRP) 31
Some Process control service 266
source of truth 427
spaghetti integration 10, 11
SQL injection (SQLi) 336
stability 34
stack 56, 57, 333
Staged Event-Driven Architecture (SEDA) 101, 126
staging 100
atomic unit of work 101
cooperative programming 101
stale-while-revalidate caching 315
Stale-While-Revalidate strategy 90
static analysis 384
story backlog 370
story mapping 371, 372
story planning 372
Strangler pattern 236, 423
dissecting 423, 424
event-first migration 424
headless mode 425
retirement 425
stream-first event sourcing 119, 120

stream processing 125, 126

subject-specific key 350

Substitution principle 266, 423, 430

subsystem bulkheads creating 48

external domain events 49, 50

separate cloud accounts 48, 49

synchronous contract 388

synthetics 393, 394, 411

system, into autonomous subsystems 44

by actor 44, 45

by business capability 46

by business unit 45

by data life cycle 46, 47

by legacy system 47

system event 406

tracking 405, 406

system of record 427

systemwide transaction log 427

T

Task BFF service 209, 266

task branch workflow 373

canary deployment 376

draft pull request (PR), creating 375

master branch, merging 375
new branch, creating 374
review 375
task roadmap 373
team capacity-driven funding 422, 423
team metrics harnessing 64, 65
teams seeding and splitting 420
technical debit 363
tests 56
third-party systems asynchronous request response 227, 228
egress, API call 226
ingress, webhook 227
integrating with 225
synchronous ESG/BFF hybrid 229
throttling 403
throughput optimization 136
asynchronous non-blocking I/O 137, 138
batching 143, 144
batch size function parameter 137
grouping 144
multiplexing 139, 140
pipeline patterns 141, 142
pipelines 139-141
sharding 143

tier innovation 67, 68

build-time, versus runtime rendering 69

client-side, versus server-side rendering 68

web, versus mobile 70

time to live (TTL) 22, 167, 197, 263

Total Cost of Ownership (TCO) 401

traditional health checks 294, 295

traditional testing method 8

transaction log 156, 157

transparency, offline-first inbox 88

outbox 88

status indicators 87

trigger function 53, 54, 61, 95, 199, 223, 365

trilateral API 58

API Gateway 58

events 58

trust gaining 417, 418

U

ubiquitous language 30

Unified Modeling Language (UML) 29

unit of work (uow) 128

unit testing 383

coverage 383

mock object 384
static analysis 384
Universally Unique Identifier (UUID) 169, 260
upstream subsystem egress flow 230, 232
USE method 402
user activities 192
autonomous service 195
decoupled service 194
frontend team 194
resilient service 194
single user activity 192, 193
user experience (UX) 17
user-interface-first (UI-first) 372

V

VCR test double 385-387
vision establishing 418

W

waterfall method 5
Web Application Firewall (WAF) 336
webpack 77
work metrics 407
alerting 406

work metrics, alerting anomaly detection 409

BFF service metrics 408

domain event metrics 408, 409

RED method 407

Z

zero-downtime deployment 364

backend 365-368

consumers 366

data store 367, 368

frontend 365

producers 366

Robustness principle 364, 365

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don’t worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don’t stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/9781803235448

2. Submit your proof of purchase
3. That’s it! We’ll send your free PDF and other benefits to your email directly