O'REILLY®

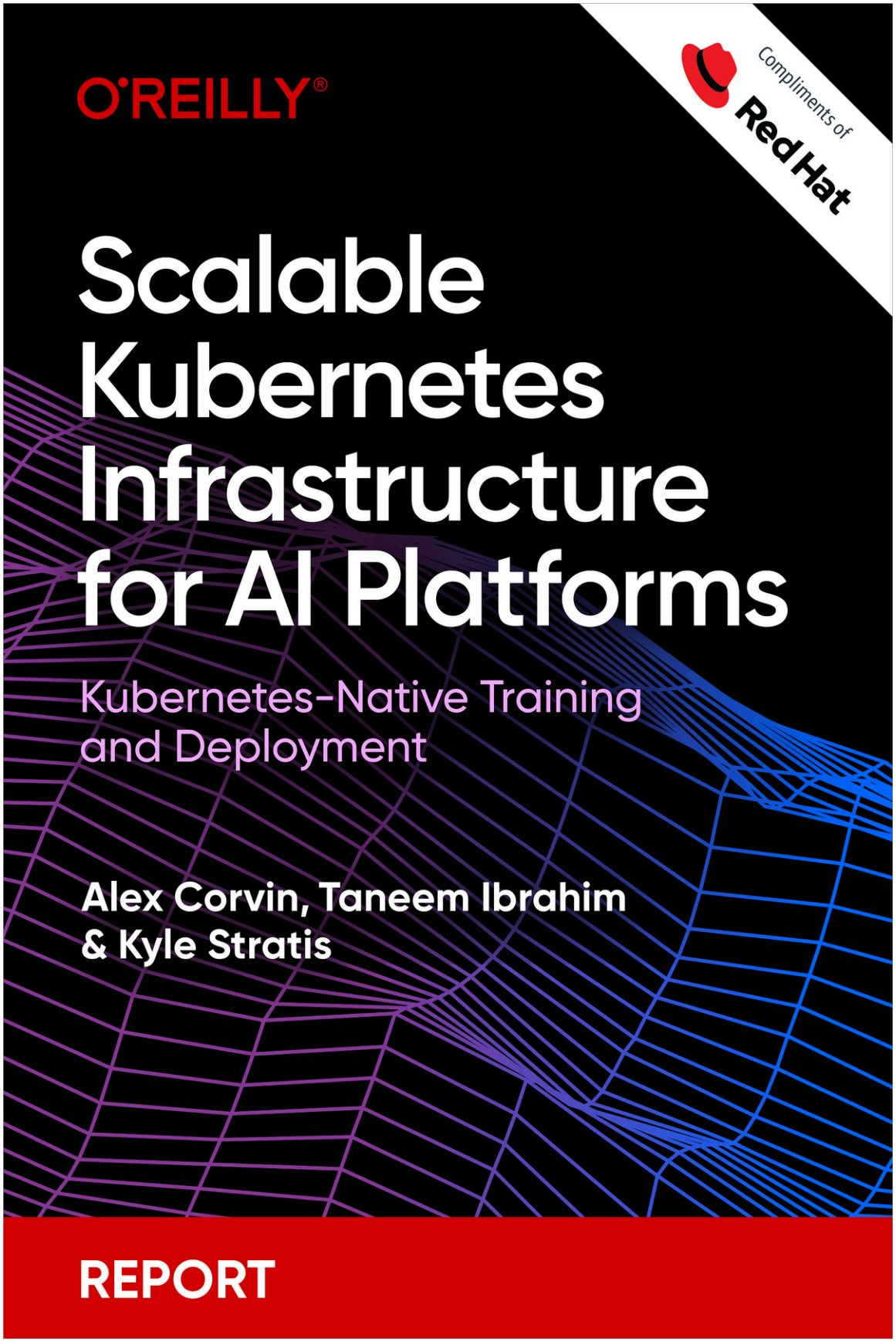
Compliments of Red Hat

Scalable Kubernetes Infrastructure for AI Platforms

Kubernetes-Native Training and Deployment

Alex Corvin, Taneem Ibrahim & Kyle Stratis

REPORT


O'REILLY®
Compliments of
Red Hat
Scalable Kubernetes Infrastructure for AI Platforms
Kubernetes-Native Training and Deployment
Alex Corvin, Taneem Ibrahim & Kyle Stratis
REPORT

Red Hat

Scalable Kubernetes Infrastructure for AI Platforms

Kubernetes-Native Training and Deployment

Alex Corvin, Taneem Ibrahim, and Kyle Stratis

Scalable Kubernetes Infrastructure for AI Platforms

by Alex Corvin, Taneem Ibrahim, and Kyle Stratis

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

- Acquisitions Editor: Nicole Butterfield
- Development Editor: Jill Leonard
- Production Editor: Christopher Faucher
- Copyeditor: nSight, Inc.
- Proofreader: O’Reilly Media, Inc.

- Interior Designer: David Futato
- Cover Designer: Susan Brown
- Illustrator: Kate Dullea

- February 2025: First Edition

Revision History for the First Edition

- 2025-02-13: First Release

code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Red Hat. See our statement of editorial independence.

979-8-341-60816-0

[LSI]

Chapter 1. Introduction

AI, especially generative AI, has followed a similar adoption cycle to many new technologies. Those organizations producing cutting-edge technologies and those in or adjacent to the tech space typically have been quicker to adopt this technology, while other industries have had a slower rate of adoption. However, more recent advancements have helped business leaders realize that they must figure out how to leverage generative AI for their businesses or risk being left behind by their competitors.

Now, enterprises are expending more resources to leverage AI for their businesses. This often takes the form of teams of data scientists implementing proofs of concept (POCs) of AI-based applications for their businesses.

A particularly common proof of concept project in the enterprise is building chatbots. Typically, these projects make use of retrieval-augmented generation (RAG), combining proprietary data with off-the-shelf large language models (LLMs) to give the chatbot expertise on a specific problem domain.

But these enterprises are often facing a difficult challenge: they have POCs of AI-based applications for their businesses, but they can't move them into production. In fact, the vast majority of these projects never make it to production.

To improve the success rate of these POCs and improve the return on investment of AI initiatives in the enterprise, businesses must develop a better understanding of the challenges that arise when running AI-enabled applications in production. With this understanding, leaders will be better able to architect solutions for promoting POCs into production and managing their product lifecycles.

Addressing and overcoming these challenges is at the core of the relatively recent discipline of *machine learning operations (MLOps)*. This publication will walk you through why this is a critical next step and how to leverage MLOps on Kubernetes.

In this report, we'll unpack four fundamentals of building AI-powered applications:

- Training models in the experimental phase
- Making model creation repeatable and declarative
- Operating models in production as a part of AI-powered applications

- Ensuring that models you create are trustworthy and built responsibly

This report will take a Kubernetes-centric approach, highlighting projects that are built to be Kubernetes native and when used together allow you to apply MLOps principles and practices to building AI-powered applications.

What Is MLOps?

MLOps has its origin in the world of *DevOps*, a best-practice development model that seeks to deliver high-quality software to production quickly. It seeks to do this by bringing development and operations roles closer together. This fosters collaboration and shared knowledge across the software development and production lifecycles while bringing awareness of production issues to the teams and individuals best equipped to solve them. This approach requires that developers concern themselves with how the software they write performs in production and that they're actively involved in operating that software in production as well.

With the proliferation of AI/ML and an ever-increasing number of models being created, MLOps has emerged as a new paradigm for delivering high-quality models to production

quickly, applying DevOps principles to AI models and AI-powered applications instead of traditional software applications. However, MLOps doesn't just apply DevOps principles to the AI development lifecycle but builds upon them to define foundational best practices for building and running AI-powered applications. A team implementing MLOps practices should adhere to the following core principles, which are expanded upon and explained in-depth in the book *Designing Machine Learning Systems* by Chip Huyen (O'Reilly, 2022):

Continuous integration and delivery (CI/CD)

A robust suite of CI/CD automation tools to repeatably build, test, and deploy AI-powered applications.

Exploratory workflow orchestration

A robust data science workflow orchestration tool to automate the end-to-end model development lifecycle from data preparation through model training, tuning, and evaluation.

Reproducible artifacts

Artifacts from a given version of an intelligent application must be made reproducible, and all components used to

create these artifacts must be versioned and well-documented.

Cross-team collaboration

Building AI-powered applications requires strong collaboration between multiple roles consisting of, at a minimum, data engineers, data scientists, application developers, and operations teams. MLOps emphasizes close communication and collaboration between these groups.

Model and data lineage

Model and data lineage along with other key metadata for an intelligent application must be well tracked, especially for the purpose of building trust in AI applications but also for debugging and explainability.

Monitoring

MLOps requires that AI-powered applications be monitored across their production lifecycle. In addition to traditional application monitoring, AI-powered applications must be monitored for data distribution drifts, model degradation, bias, compliance, and more. Because many models also use expensive specialized hardware, such as GPU clusters, monitoring for the

efficient use of this hardware is critically important as well.

Iteration-supporting process

MLOps processes must allow for frequent iterations throughout the development and production lifecycles of an intelligent application. Data scientists must be able to train a model, evaluate its performance, and quickly retrain the model based on results of the evaluation. Similarly, models must be periodically retrained after they are released to production in order to incorporate new data in their training sets, as new data may diverge from the original training data. This divergence can be caught as it happens by adhering to the previously mentioned monitoring principle.

Now that we understand the foundational principles, let's consider the AI development lifecycle. While this lifecycle can take many forms, all tend to roughly follow this pattern (illustrated in Figure 1-1):

Project initiation

Business stakeholders, application developers, data scientists, and data engineers collaborate to identify the desired business outcome for the intelligent application,

the raw data that can be used to build the model that will achieve this outcome, and the architecture for how the resulting AI model will be integrated into a software application to deliver the desired solution.

Data preparation

Data engineers and data scientists produce the necessary training and tuning data artifacts upon which to build the model. These artifacts do not need to be static: they could be real-time data or database entries that are constantly updating from data ingestion pipelines, for example.

Model experimentation

Data scientists consume the prepared feature data to create a sufficiently performant model, frequently iterating on different model architectures, training hyperparameters, and feature data combinations.

Application integration

Application developers work closely with data scientists to integrate a trained model into application code, which will consume the trained model via an API.

Production service

The application is promoted to production, where it adds value and is continuously iterated upon to improve its performance and add new features.

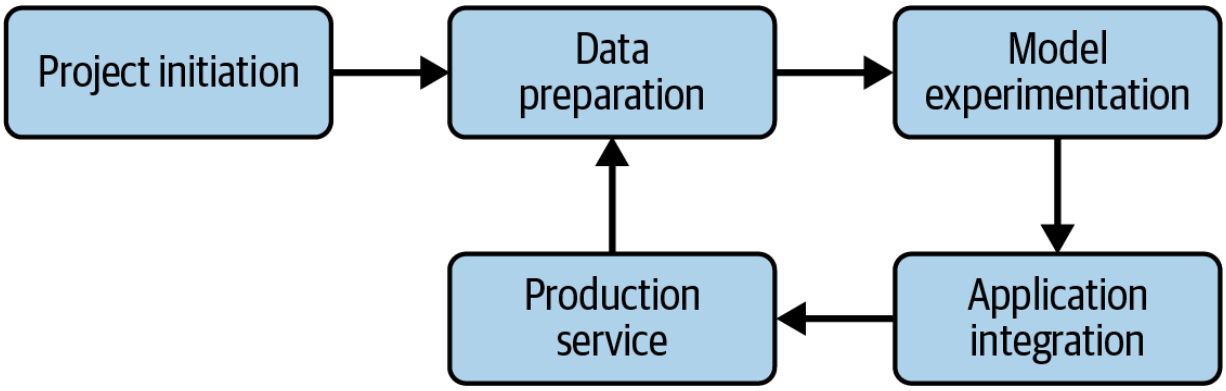


Figure 1-1. This flowchart demonstrates the iterative nature of the AI development lifecycle

Given the strong demand for a platform that enables this lifecycle and MLOps best practices, it isn't surprising that a dizzying number of AI development platforms exist on the market today. This report will dive into the open source Kubernetes container orchestration platform, highlighting how it can be used across the AI development lifecycle to apply the foundational MLOps principles to your workflows.

Why Use Kubernetes for Your MLOps

Platform?

Kubernetes has many strengths, which make it an excellent platform on which to build and run AI-powered applications while adhering to MLOps principles. Because Kubernetes applications are written in a declarative manner, it allows teams to consistently produce repeatable results when building AI models. This, combined with built-in robust GitOps tooling, makes it very easy to version control model training artifacts.

Creating models requires access to specialized hardware, and how that hardware is used is often unpredictable and bursty in nature. Kubernetes is able to abstract away the process of provisioning hardware resources such that a data scientist can focus on developing the model instead of on configuring the hardware environment. On the production side, running AI-powered applications requires separately scaling different parts of the application, including the compute resources serving the backend model and frontend APIs that provide a user access to the model. Because Kubernetes abstracts away hardware provisioning, scaling different pieces of the deployed application becomes easier.

In addition to manual scaling, Kubernetes allows users to automatically scale highly specialized and expensive compute resources. This fine-grained resource management is critical for properly managing costs.

Another consideration is specialized hardware such as accelerators, particularly in large-scale training and tuning jobs, which can be quite fragile. In a model training or fine-tuning job that requires multiple days to execute, a hardware failure that forces you to restart training can be quite costly. Kubernetes has self-healing features, which, coupled with checkpointing support in common training libraries, eliminate this problem, making Kubernetes a robust fault-tolerant platform.

While many AI development platforms are tied to a specific cloud platform, Kubernetes is able to run anywhere you need it. This includes cloud providers, private datacenters, edge locations, and hybrid configurations, which allow Kubernetes to serve as the single deployment platform upon which to build your applications.

On the monitoring side, Kubernetes integrates with several monitoring tools such as Prometheus, DataDog, and Grafana, which can help track performance and resource usage of

models. This is especially important for LLMs due to their size and cost to operate. These deep integrations provide MLOps administrators with proactive monitoring and alerts to ensure that models run optimally for critical AI workloads.

Finally, rolling out updates to models, especially LLMs, can be a difficult and costly practice. Kubernetes simplifies this process with features like rolling updates (which pushes updates incrementally) and canary deployments, helping to minimize the downtime of these models.

Chapter 2. Model Development on Kubernetes

In this chapter, we will provide an overview of prevailing technologies and techniques for developing machine learning models using Kubernetes as a compute platform. While we will focus on specific techniques relevant to large language models (LLMs) and generative AI, many of the techniques we discuss will apply to traditional predictive models and other architectures as well.

Historically, models have required extensive data preparation to curate high-quality, labeled datasets that sufficiently capture the problem domain. Creating these datasets was very labor-intensive and expensive. More recently, advances in computational power, improved algorithms for distributing training across compute resources, and widespread open access to training data have all paved the way for extremely powerful general-purpose models to be built without heavy data curation.

Generally, foundation LLMs are created via self-supervised learning, a type of unsupervised learning, on extremely large, unlabeled datasets. For LLMs, this results in a model that

understands patterns in human language and can predict the most likely output that should follow a given input. These foundational models exhibit usefulness across a wide breadth of tasks, but practitioners often need to adapt these pretrained base models to some specific use case.

There are several prevailing techniques for adapting these foundational models, which differ from each other in their intended use cases, ease of implementation, and costs of implementation. Collectively, we will refer to these approaches as *model customization techniques*.

Overview of LLM Customization Techniques

The LLM customization space, like much of generative AI, is evolving rapidly with new techniques being invented regularly. In general, customization is achieved through one or more of the following fundamental techniques:

- Customizing an existing model's output by leaving the model unchanged but carefully constructing the input to get a desired result. Examples of this include *prompt engineering* and *retrieval-augmented generation* (RAG).

- Combining individual models to achieve an output that is more desirable than that from a single model. One example of this is the *mixture-of-agents* approach.
- Retraining an existing model using curated data specific to a given task. This is *fine-tuning*.

Novel model customization techniques are likely to be achieved through new algorithms for implementing these fundamental techniques more efficiently or through creatively combining these techniques, as in the case of *retrieval-augmented fine-tuning* (RAFT).

In the rest of this section, we will provide a primer on two of the most prominent approaches (as of this writing) to model customization: RAG and fine-tuning.

Retrieval-Augmented Generation

A fundamental limitation of pretrained foundation models is that they possess "knowledge" only of the data that they were trained on. If you ask a model about a piece of data that it was not trained on, it will fail to give the desired answer. RAG is a technique that extends an existing model's knowledge by passing relevant contextual data as input to the model at query time.

So how does RAG work? Generally, when a user queries a model, a database (typically a vector database) is queried for information relevant to the input query. The RAG system parses the results and uses an algorithm like cosine similarity to choose the results most relevant to the query. Once those are chosen, they are added to the original query as contextual information and sent on to the model in a format along the lines of "using information found only in this input document, answer this question for me." Figure 2-1 illustrates a hypothetical RAG system.

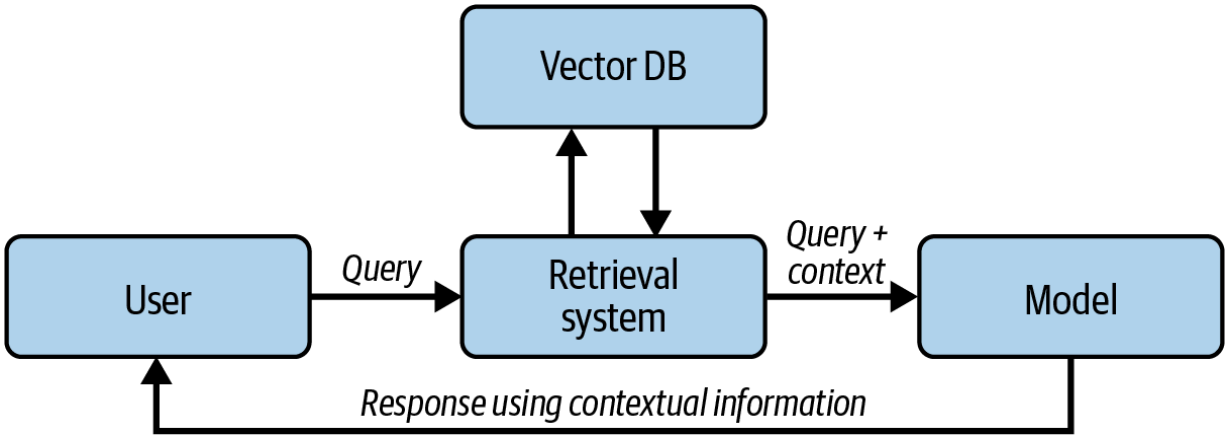


Figure 2-1. An illustration of a generalized RAG system showing the interactions between the user, retrieval system, vector database, and model

The chosen retrieval and ranking algorithm is critically important to the performance of the RAG system. If no relevant contextual data is retrieved by the system, the model will lack the knowledge needed to give the desired answer to the user.

Even though RAG requires a retrieval system and additional data storage between the user and the model, it has a number of benefits. Because RAG supplements the model's knowledge at runtime, it requires less knowledge to be baked into the model and opens up the possibility of using a smaller model that is cheaper to serve to users while simultaneously allowing users to incorporate rapidly changing data like stock prices on the fly. RAG can also reduce the time to achieve value with an LLM, because it doesn't require a lengthy retraining process to work.

On the other hand, the knowledge given to a model via RAG is transient, and only exists for a single query. You also have to carefully craft the input prompt to get the kind of output you're interested in. However, this sort of customization also has its limits. If you want to make knowledge changes persistent or fully customize the format of the model's output, retraining the foundation model is required.

Model Fine-Tuning

Training a foundation model is notoriously expensive and time-consuming, which isn't an option for even the largest enterprises. Instead, we can make use of a technique called fine-tuning. With fine-tuning, you create a high-quality, labeled dataset that is specific to your domain-specific task, knowledge,

or desired output format. You can then use that dataset to adjust a pretrained model in a fraction of the time and with a fraction of the data that would be required for training from scratch.

The fine-tuned model will then have your desired knowledge and behavior baked in, allowing your production architecture to avoid the complexities required by techniques like RAG. However, fine-tuning requires knowing how to train a model, the time to curate a training dataset large enough to influence a model, and the sometimes high compute cost to perform the training itself.

A number of techniques exist to optimize the compute cost of fine-tuning, such as parameter-efficient fine-tuning (PEFT) and low-rank adaptation (LoRA). Both of these techniques work by training only a subset of the pretrained model's weights and biases.

While this is complex, there are many tools and entire platforms available to help with training and fine-tuning models, such as InstructLab and Hugging Face's sft_trainer, with many of them available within the open source Kubernetes ecosystem.

Kubernetes-Native Model Training Tools

While many training tools and platforms are available, at a fundamental level they all provide easy access to the compute power necessary to train and fine-tune models. When evaluating a training tool or platform, the following requirements should be considered:

- Integration with the training framework(s) (distributed or otherwise) that data scientists or data science teams use and are comfortable with (e.g., PyTorch, TensorFlow, etc.).
- Support for training/fine-tuning algorithms that your team wants to use.
- Access to hardware optimizers, such as accelerators (e.g., GPUs), specialized network devices, and specialized storage providers with multi-write-capable storage.
- Integrations with the development environments data scientists or data science teams are already using. A tool that effectively abstracts away Kubernetes so that the data scientist or team doesn't need to manage it is ideal.

In the following subsections, we will explore open source tools that meet these requirements and have strong community

adoption.

Ray

Ray is a framework that enables users to scale their training and fine-tuning processes up from single machines to clusters of machines, and can run natively on Kubernetes via the KubeRay operator. It seamlessly integrates with PyTorch and other frameworks via Ray Train and has extensive support for accelerators. It also comes with a dashboard that provides key monitoring information to end users.

NOTE

An operator is an extension to Kubernetes that helps to manage Kubernetes applications by using custom resources to automate the application's lifecycle.

Ray's biggest strength is its ease of adoption by data scientists who don't know Kubernetes well, but it comes with the downside of increased overhead through the management of Ray clusters when compared to options that have a more "raw" interface to Kubernetes. It also doesn't always scale well to extremely large-scale training jobs.

Kubeflow Training Operator

Kubeflow is a community-managed open source ecosystem of Kubernetes components that support the full AI lifecycle. A part of that ecosystem, the Kubeflow Training Operator (KFTO) is a Kubernetes-native operator that allows users to use Kubernetes for distributed training and fine-tuning of large models. Its software development kit (SDK) allows for easy integration into existing environments and code and has extensive support for common frameworks like PyTorch.

KFTO accelerator support is tied to the chosen training framework, so it supports anything that the training framework and Kubernetes support and can scale to any level that the framework and Kubernetes are capable of scaling to. Unlike Ray, KFTO is a thin layer on top of the underlying Kubernetes objects, which introduces very little compute overhead. The flipside to that, though, is that more of the Kubernetes details are exposed to the user, which may be confusing for data scientists and developers who do not need to know these details.

Native Training Framework Integration

with Kubernetes

Most training frameworks have framework-specific tooling for integrating with Kubernetes to provide computational resources. PyTorch, for example, has a universal job launcher called TorchX that includes Kubernetes support via its scheduler. While this kind of solution is the most lightweight and is the easiest for data scientists to adopt, it is less declarative and thus doesn't lend itself as well to administration by MLOps teams.

Another potential downside is that these tools are framework-specific, so usage won't necessarily scale in large organizations with several data science teams using different frameworks. These native integrations are best suited for small teams of data scientists during experimentation phases.

NOTE

Typically, once a model is trained or fine-tuned, you will want to evaluate its performance. Many existing model evaluation tools that data scientists use outside of Kubernetes can also be used when Kubernetes is used as a training platform.

Managing Compute Resources for Training

While the tools described in the previous section allow you to train and fine-tune across many computational resources, this often requires extensive and costly hardware resources. Enterprises must pay particular attention to managing the cost incurred during training or fine-tuning. A robust management system should be able to do the following:

- Facilitate the creation of job queues so that requests for compute hardware get serviced as soon as the hardware becomes available.
- Assign resource quotas to groups of users in order to constrain how many resources a given group can consume.
- Share resource quotas between groups when individual groups need to burst and there are free resources.
- Manage request priorities for resources and priority-based job preemption.
- Provide auditability and reporting on resource management at the model, job, and team levels.
- Allow all of these functions to be centrally managed by IT teams while maintaining transparency for users.

There are currently two major open source projects in this space: Kueue and Volcano. Both projects are Kubernetes native and have strong community adoption. They also have support for managing resources of various types, like Ray clusters, KFTO jobs, and PyTorch training jobs.

While these projects offer similar functionality, they do have some key differences. Kueue is an official Kubernetes special interest group project and is thus “blessed” by the wider Kubernetes community. It is based on the design principle of delegating functionality to existing Kubernetes components when applicable, and because of this, Kueue is fairly lightweight.

Volcano, on the other hand, replicates some existing Kubernetes functionality, giving it more overhead but allowing it to be a more holistic and better-integrated solution. It is also more mature than Kueue and as of this writing offers more capabilities.

Once a data science team has a model and training procedure it is ready to send to production, it will be necessary to periodically retrain the model while keeping track of the datasets that went into each new version of the model. In Chapter 3, we will discuss why periodic retraining, model

versioning, and dataset versioning are necessary along with tools to help with these production workflows.

Chapter 3. Making Training Repeatable

In Chapter 2, you learned about techniques for customizing a model, including fine-tuning, a special case of model training. Once you've fine-tuned or trained your model for the first time, you might be tempted to think that you are done with model development and all you have left is to evaluate and deploy your model.

You'd be half right. But because the data that informs the model will likely change over time, the model must be regularly retrained throughout its lifetime to ensure that it can continue to deliver value. In this chapter, you will dive into the AI model lifecycle, learning how to track model versions, automate model training, and implement GitOps for model training pipelines.

Retraining and the Model Development Lifecycle

The world changes. The data that we use to describe the real world, then, must change too. Consequently, if the data changes,

then any models that attempt to model a problem in the real world must also change.

Since models in production are static, over time the input data that a given model will process in production for inference requests will differ from the data that the model was trained on. This variability can come from any number of sources, such as changing user behavior over time, seasonality effects in the data, changes to the input data format, etc. The phenomenon is called *data drift*.

Data drift isn't the only reason to retrain a model, however. Retraining is a good option when a metric that is monitored in production (such as accuracy, model responsiveness, compute resources, etc.) falls outside its optimal range.

When and how often a model should be retrained are two key considerations, and they depend heavily on your use case and training data. There are two main choices here: either regularly retrain the model on some fixed cadence, or retrain the model on demand.

Retraining a model at a fixed cadence can be costly if the cadence is rapid or if the retraining doesn't actually show any improvement. This method assumes that the data changes

according to some predictable pattern that can be detected at a chosen retraining cadence. If retraining doesn't show any improvement, it's possible that the data isn't changing in such a way to be captured by the regular cadence.

The other option is to retrain your model on demand. This ensures that models are not retrained unnecessarily, but it requires reliable monitoring of the performance of the current version of the model (discussed in Chapter 4) and well-defined thresholds for when the performance has sufficiently degraded.

The full lifecycle of a model, then, looks something like Figure 3-1.

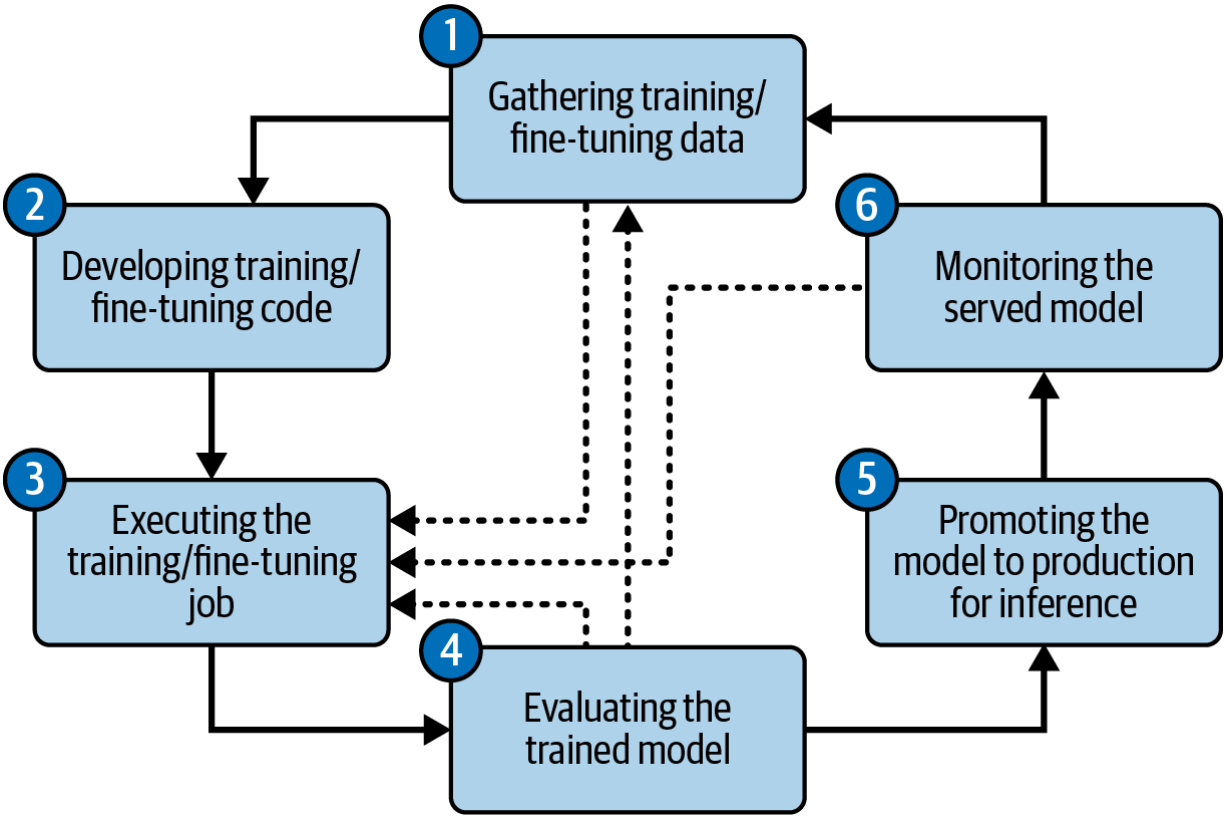


Figure 3-1. The full lifecycle of an AI model

In practice, this is a never-ending cycle, which repeats most often from the evaluation and monitoring stages, where unacceptable performance is typically discovered.

Where Figure 1-1 was focused on the model development cycle (stage 2), Figure 3-1 zooms out to view the entire lifecycle of a model. The lifecycle starts with gathering data, continues through developing training code, executing the training job, evaluating the trained model, promoting the model to production, and monitoring the served model.

Crucially, this lifecycle can repeat at any stage (with the exception of training code development, which typically remains static), most commonly after the evaluation and monitoring stages (4 and 6). From those, it's common to discover that either the training job needs to be run again, or that more or higher quality data needs to be collected. If more data is to be collected, typically the existing training code can be used as-is. In this case, stage 3 would follow stage 1.

Tracking Model Versions

While they iterate on developing a model, data scientists will run many experiments by varying the dataset they use to create the model, the model's architecture, the hyperparameters used for training the model, and more. Even after the initial handoff of the model to production, future retraining cycles of the model will yield new variants.

Enterprises need robust solutions for tracking all of these versions of a given model. While version control systems have been table stakes for traditional software projects for decades now, model version control systems are still in their infancy despite the heavy reliance on models by enterprises of all sizes.

Model version tracking benefits the whole enterprise, from unlocking the ability of data scientists to recreate previous experiments, share their experiments with colleagues, and revert to a model from a previous experiment, to enabling model auditing and ensuring responsible AI use. For example, in order to explain a given model's results and how it was created, it is necessary to know which version of a model was served in production at what time and how that model was created, including the data sources that went into the model.

More and more often, enterprises use a centralized model registry throughout the model's lifecycle, and this is rapidly becoming a recognized best practice. During model training and development, model training code should integrate with the model registry to register each subsequent version of the model as a distinct model artifact. Alongside the model artifact, model training code should register metadata about the training data and code that was used to create the model. The model registry should also be integrated at model deployment and monitoring stages, which will be discussed in Chapter 4.

Many training platforms that are available today offer model registries as part of their product lineup. It is important to choose a platform with a strong model registry that is well integrated into the platform's distributed training engine.

The Kubeflow project offers a powerful integrated solution, where the Kubeflow Training Operator, Kubeflow Pipelines, Katib, and the Kubeflow Model Registry can be used together to track training results in a central database. Similarly, the MLflow project offers a robust model registry solution that can be combined with MLflow Runs and Experiments to streamline tracking of model versions.

Today, proper model versioning tends to require behavior change by data scientists to deliberately integrate model version tracking into their training code and workflows. As projects like MLflow and Kubeflow evolve and become better integrated with frameworks like PyTorch, you can expect model version tracking capabilities to integrate seamlessly out of the box.

Automating Model Training

Training a model is just like any other regularly repeated computing activity: it must be automated. Failure to do so leads to several negative outcomes:

- Wasted human resources by having to manually rerun training

- Unpredictability in model retraining cadence
- Inconsistency in the model's performance through discrepancies (deliberate or accidental) in the model training process

Initial model development, then, should not be considered complete until the training process is automated end to end. A fully featured automated training process should include at the very least:

- Input parameters that specify any variables that typically need to be tweaked (e.g., a version identifier for the training code or training data, or *hyperparameters*—parameters that define how training is done—for the training job)
- Any necessary data preparation or preprocessing to collect data from any storage locations and prepare it for training
- Fetching and executing the training job
- Storing the model and related artifacts in a chosen storage endpoint
- Registering the model and related artifacts in the model registry
- Evaluating the performance of the trained model

Another feature that is nice to have but by no means essential is an automated process that promotes a trained model to

production or any other post-training steps if the model meets some minimum performance thresholds.

WARNING

For many teams, promoting a model to production without any human oversight may not be appropriate. When evaluating a tool that has this feature, be sure to weigh the trade-offs of not having human review of a trained model's metrics against keeping a human in the loop.

This process (and the software that implements it) is often referred to as a *pipeline* or *workflow*. These pipelines are usually authored by data scientists, data engineers, machine learning engineers, and/or MLOps teams, with Python being the prevailing language of choice.

For enterprises that have adopted Kubernetes as their model development and serving platform, we strongly recommend adopting a pipeline engine that is Kubernetes native and thus is able to leverage the existing Kubernetes infrastructure, integrating seamlessly with the overall MLOps infrastructure in use.

There are three major open source pipeline engines that have strong community adoption and should be considered for your training infrastructure:

Airflow

Airflow is typically preferred by data scientists and tends to present the cleanest experience for authoring pipelines—or directed acyclic graphs (DAGs) in Airflow's parlance)—but Airflow is not strictly Kubernetes native, which presents challenges when operationalizing it at scale on Kubernetes.

Kubeflow Pipelines

A part of the broader Kubeflow project, Kubeflow Pipelines is Kubernetes native at its core, making it more customizable, scalable, and well suited to enterprises with large or multiple data science teams wishing to share Kubernetes infrastructure, or with central IT/MLOps teams managing consistent infrastructure across teams.

MLflow

MLflow excels at model version tracking, making it easy for data scientists to adopt and track multiple versions of their models over time. However, MLflow requires more effort by operations teams to deploy and scale on Kubernetes.

TIP

Do you want to see what a continuous model training pipeline looks like in action? Red Hat offers an MLOps lab exercise that helps you build one yourself.

GitOps for Model Training Pipelines

Model training pipelines are like code and should be treated accordingly. Pipeline definitions should be stored in source control and versioned, just like traditional application code. And like traditional application code, authors of pipeline definitions should follow a robust peer review process when making changes to the definitions.

Production training runs, then, should run from clean versions of these pipelines, pulled from a well-defined version (such as a branch or tag) of the pipeline in version control. This is GitOps in a nutshell. GitOps is a recent development in traditional software operations whereby applications are cleanly deployed from version control and continuously reconciled to ensure that the deployed application matches the desired state in version control. When teams wish to change the state of the application in production, they do so by changing the application's definition in version control.

Kubernetes' declarative approach to deploying applications, along with the reconciliation loop that keeps Kubernetes applications in their desired state, make Kubernetes an ideal platform for managing pipelines with GitOps.

For managing applications on Kubernetes, one of the most popular projects is Argo CD. Argo CD is a continuous delivery tool for Kubernetes that allows users to implement GitOps principles into their workflow. While deploying pipeline definitions with Argo CD typically requires the development of custom code to convert pipeline definitions from version control into runnable training jobs, this is an area of rapid innovation, especially from the Kubeflow project, to allow for simpler pipeline definition management.

Now that you've learned how to reliably retrain a model, keep track of its versions (and that of the data and training pipelines), and build a more robust open source MLOps infrastructure, you are prepared to move on to the next step of the AI model lifecycle: deployment and monitoring.

Chapter 4. Model Deployment and Monitoring

In the previous chapters, you learned about model customization techniques, including fine-tuning and training, and about making training and evaluation repeatable. Once you've achieved the results you are looking for with your model, it's time to deploy your model to production.

This chapter will prepare you for model deployment and serving by giving you an overview of the major technologies and techniques used with Kubernetes to deploy and monitor machine learning models. While we will focus on specific techniques relevant to large language models (LLMs) and generative AI, much of this chapter will also apply to traditional machine learning models.

Overview of LLM Serving

Model serving is the act of processing inference requests in real time, which requires deploying an already trained model to some location suitable for receiving these requests. At a high level, model serving involves packaging the model, deploying it

on hardware accelerators like GPUs or CPUS, exposing APIs for users to query the model, and enabling metrics for monitoring and alerting. The components of a model-serving system include model-serving platforms, model-serving runtimes, and metric gathering and monitoring systems. Typically, an API gateway and load balancer to handle bursts of traffic for model queries is also included.

The model-serving platform component retrieves the model from storage (such as Amazon S3 or a local persistent volume) and then performs various preprocessing tasks such as changing model formats or postprocessing steps such as gathering metrics. It incorporates a model-serving runtime in order to help it serve the model. It also exposes a REST or gRPC API so that the models can be interacted with by users while providing model access security through the gateway and load balancer.

The model-serving runtime component loads the model into the GPU or CPU memory, deserializes any incoming query or prompt from its over-the-wire representation, converts it into a format suitable for the model, and then executes the inference on the model to retrieve a response. This response is typically serialized into a JSON object or other format and sent back to the calling application.

The metrics and monitoring components typically aggregate the request metrics such as the request time, error codes, token count, tracing, and more, storing them to a metrics server like Prometheus. These metrics allow an MLOps practitioner to ensure the health and performance of the models in production and diagnose any issues that may come up via alerting.

Although the majority of development and compute time is spent on training and tuning a model, nearly 90% of a model's lifecycle is spent serving, which is why optimizing serving can be pivotal to delivering business value from the model. In the next sections, you will learn about each of these essential components using Kubernetes-specific tools that will help you scale up your generative AI inference workloads.

Using a Model-Serving Platform

A model-serving platform is the core component of any inference system, managing model deployment and scaling according to the volume of incoming inference requests. There are currently many platforms available for serving models on Kubernetes. Their purpose is to simplify and scale the model deployment and inference serving processes.

In order to serve an LLM in a scalable fashion on Kubernetes, a serving platform should meet these requirements:

- Support for different types of model architecture
- Extensible by adding new model architectures
- Support for generating embeddings for different modalities, such as text or image
- Support for multiple modalities in inference
- Support for chaining inference across multiple models (*model composition*)
- A wide range of hardware accelerator support
- Integration with standard Kubernetes APIs and tools
- Robust support for different model artifact formats
- Support for a wide range of storage technologies
- Integration with API gateways
- Ability to deploy models in an A/B or canary rollout fashion
- Provide flexible integration options with pre- and postprocessing systems
- Support for automatically scaling inference infrastructure
- Integration of model monitoring solutions

One of the most popular tools for deploying LLMs with Kubernetes is KServe. KServe is a controller for Kubernetes that enables Kubernetes to serve both predictive and generative AI models along with maintaining inference request pre- and

postprocessing pipelines. Not only does it meet the previously mentioned requirements, but KServe also provides several benefits that have helped its widespread adoption in the enterprise:

- An active and thriving open source community
- Support for traffic routing and autoscaling, including scaling to zero
- Support for both batch and real-time inference workloads
- Support for both predictive and generative AI inference with a standard protocol, the Open Inference Protocol

Like other controllers, KServe is composed of a set of custom resources, which are extensions to the base Kubernetes API. One of these is the ServingRuntime. This is essentially a deployment template that defines the environment from which models will be served. KServe comes with a number of ServingRuntimes available out of the box, but others can be easily added to the system. Each one defines things like the container image to be used for the runtime and the model formats that the ServingRuntime supports, and can be further customized via environment variables set in the container. This allows users to easily add support for new model architectures.

At the core of KServe is the InferenceService. This is a custom resource definition where you define predictors, storage locations, model format, canaries for gradual deployment, deployment mode, and anything else required to serve your model. Models are typically initialized from cloud storage like Amazon S3 buckets, but it is also possible to use OCI-compliant containers as an alternative to cloud storage with KServe "Modelcars".

To use this, an OCI-compliant container image must be created and then added to a container registry like Quay. When you deploy with KServe, you can then reference the repository holding the container. Because a Kubernetes cluster keeps a cache of downloaded container images, the model doesn't need to be downloaded multiple times, which can reduce startup time while still reducing overall disk usage.

NOTE

KServe offers three deployment modes: `RawDeployment` mode uses a standard Kubernetes deployment and ingress gateway (an API gateway for routing inbound requests only); serverless mode uses Knative objects to enable serverless deployment; ModelMesh allows for multiple models to be deployed in a pod to scale smaller models with fewer compute resources.

The serving runtime is used within an InferenceService, and the InferenceService is managed by the KServe Controller (Figure 4-1). This ensures that the deployed application state matches the definition of the InferenceService, creating the deployment for each inference endpoint and enabling features like autoscaling.

Each endpoint is composed of three components:

Predictor

This is the only required component of an endpoint. It consists of a model and model server that makes the model available at the endpoint.

Transformer

This component allows users to define both pre- and postprocessing steps as needed to manage incoming request data and outgoing inference data.

Explainer

This enables an alternate workflow that provides both predictions and model explanations. KServe provides APIs so that users can write and configure their own explanation containers.

When a user calls a KServe endpoint with `:predict` or `:explain`, that request is routed to the three components. For either call, the transformer component is the first stop for the request. If `:predict` was called by the user, the request is then routed to the predictor. If `:explain` was called by the user, then the request is routed from the transformer to the explainer component, and then the explainer calls `:predict` on the predictor component (Figure 4-1).

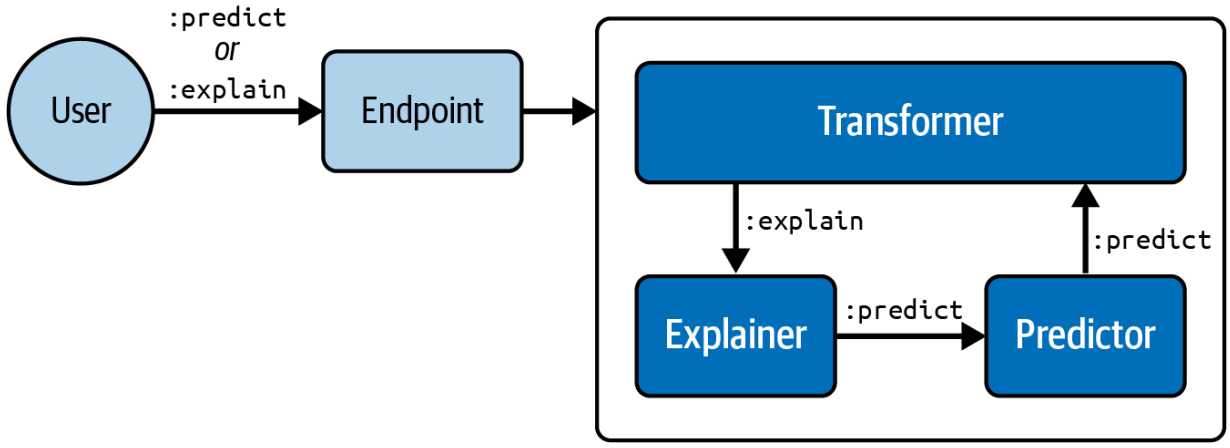


Figure 4-1. The request flow for a user calling a KServe endpoint with the `:predict` or `:explain` calls

While KServe has many features that make it a well-functioning platform for deploying machine learning models out of the box, LLMs often require some additional work.

Diving Into LLM-Serving Runtimes with vLLM

Because KServe allows you to define your own ServingRuntime resources, it is possible to use alternative model-serving runtimes with it. One such runtime is vLLM, a serving system tailored for LLMs that aims to enhance inference efficiency and scalability. It addresses the challenges of deploying LLMs by optimizing memory usage and execution speed, making it suitable for real-time applications that require high throughput and low latency.

NOTE

KServe also includes an out-of-the-box LLM runtime, the Hugging Face LLM Serving Runtime, which uses vLLM as its default backend.

---

vLLM provides a server built on FastAPI for online model serving that is compatible with the OpenAI API and also with popular machine learning frameworks like PyTorch, allowing for seamless integration with existing machine learning pipelines and facilitating the deployment of models trained with these frameworks.

vLLM also supports dynamic batching, which groups multiple inference requests into a single batch to improve processing efficiency. This is particularly beneficial in high-traffic scenarios, where it can significantly increase throughput.

The keys to vLLM's serving speed are a few core architectural features:

Paged attention

This is an algorithm that allows the storage of large continuous key-value pairs in noncontiguous blocks of memory, optimizing memory use.

Parallel execution

The architecture supports the parallel execution of model components by leveraging model parallelism and tensor slicing. This allows different parts of the model to be processed simultaneously across multiple hardware units, optimizing resource utilization and speeding up inference.

Tensor caching

vLLM has an in-memory tensor store, which caches frequently accessed tensors to avoid repeated

computations. This significantly reduces the time needed for inference by providing fast access to necessary data.

While many model-serving runtimes can be used with KServe, vLLM is a strong contender for serving LLMs due to these features. Once your model is deployed, it's imperative to be able to understand the model's performance over time and to keep track of model families and versions that you have available to your system. In the next section, you will learn about how to monitor LLMs, what metrics to monitor, and how to use registries to keep track of your deployed models.

Monitoring and Keeping Track of Your Models

Whether you have one model or hundreds in production, it is essential to monitor their performance in real time and to keep track of which models you have in production. To monitor performance, you'll have to know which metrics you should track. This is partially dependent on your use case and infrastructure, but there are also general metrics that you can track. Once you've built an understanding of metrics to track for deployed LLMs, we will move on to monitoring these metrics in KServe and then tracking your models in a registry.

LLM Metrics

LLM evaluation is a rapidly advancing field. After all, how do you tell if an LLM is doing what you want? Are you worried about *hallucinations*, or factually incorrect output? Or is machine creativity important for your use case? Or maybe you don't care as much about content but want to make sure that your model is serving inference requests at an acceptable rate and that requests are not getting stuck in queues.

For task-based metrics, such as those measuring summarization or translation, there are countless metrics available to use. You should use caution, however, since these tasks are open-ended, and it is unlikely that there is a single metric that will give you an accurate view of your model's performance. You will have to evaluate the metrics for your use case and pick a combination that accurately conveys your model's performance on its specific task. A wide variety of summarization task-specific metrics can be found in the article "LLM Evaluation for Text Summarization", published by Neptune.

Model-serving runtimes typically come with their own metrics that measure how well the server is handling requests at every stage of processing. vLLM, for instance, comes with a large metrics class that holds many different useful metrics for LLMs.

Some especially important ones are `gauge_gpu_cache_usage` and `gauge_cpu_cache_usage`, which show how much of the key-value cache mentioned earlier in this chapter is being utilized, `num_requests_waiting`, which shows how many requests are waiting to be processed, and `num_requests_running`, which shows how many requests are being processed. All of these metrics are exposed by the `/metrics` endpoint.

vLLM's metrics, as well as those exposed by KServe, can be integrated into KServe and visualized in Prometheus. In KServe, all model-serving runtimes are able to export metrics in a Prometheus-compatible format.

Prediction Logging

Prediction logging is important in traditional machine learning, and it remains important with LLMs. Sometimes this is also called *generation logging* since the result of a generation is logged. When combined with the input prompt, input and output tokens used, and other generation metrics, prediction logs become a powerful tool for auditing model usage and accuracy.

These can be stored in any existing log storage solution, and all-in-one solutions like MLflow integrate prediction logging and viewing into their platform.

Production Model Registry

In Chapter 3, we covered the overall importance of keeping a single, centralized model registry for the entire AI lifecycle, as well as why the preceding stages of the lifecycle need a registry. What we haven't covered yet, though, is what a model registry brings to production.

When getting ready to deploy a model, it's important to know which version of the model is ready for production and which version, if any, is currently deployed. This is information that you would store in the model registry, along with each model's metadata, such as evaluation results and hyperparameters, which can help with the decision to deploy or with configuring the serving environment.

Once deployed, the model registry can still provide important benefits, mostly around monitoring and observability. A registry can help users easily find model artifacts to track performance metrics for specific deployed model versions,

understand traffic patterns to those versions, get training and other details to diagnose issues found in production, and more.

NOTE

Like model registries, some groups are beginning to experiment with *prompt registries*. While these aren't currently widely available, this will be an area of innovation to watch out for, as prompt registries could provide many of these same benefits to prompts themselves.

Not only does a deployed model need metric monitoring, but it also needs safeguards and compliance built in to prevent abuse. This has become especially urgent with LLMs, due to how well they produce convincing natural language and due to the open interface with users they have compared to traditional APIs, creating a vast attack surface. This poses additional challenges for safely and responsibly serving LLMs.

Chapter 5. Responsible AI

Up until now, we focused on building a model and preparing it for deployment. Before going live, though, there are additional important considerations to cover.

Large language models (LLMs) are not just powerful from a technical standpoint but also from a societal standpoint. They're being used to amplify spam, aid scammers, and even generate plausible propaganda at scale. They are trained on massive corpuses of data, which include material we want to draw from as well as all kinds of bias, racism, sexism, and potentially illegal or harmful material (for example, early versions of ChatGPT cheerfully explained to users how to build bombs). Because of this, there has been increased emphasis on and scrutiny of ethical and responsible training and use of LLMs.

Data Safety and Transparency

Like any other machine learning model, training data is of utmost importance to LLMs. Unlike other machine learning models, however, LLMs generate text, which creates a new attack vector for bad actors. One way to combat this and help

prevent unethical data usage is to publish clear information about how an LLM was trained, what data it used, and who developed it. This information makes it possible for users, researchers, and regulators to effectively scrutinize the model's behavior and to report or manage instances of harm or misuse. There are several platforms and frameworks available to help achieve this, such as UNESCO's Global AI Ethics and Governance Observatory, with more being developed and iterated on by government groups, private think tanks, organizations building AI, and academic labs.

Because of the wide variety of possible outputs of an LLM, there are various outputs that a model could generate that could be a security vulnerability (like leaking personally identifiable information [PII]), a safety or ethical violation (such as generating racist or sexist content), or harmful information (like how to commit suicide or to build a bomb). These topics are challenging to consider, and the open-ended nature of LLMs multiplies the challenge, but it is important to consider them due to the wide deployment and novel capabilities of this technology.

As AI becomes more regulated, transparency will be key to ensuring compliance with laws and standards related to data privacy, use, and security. Transparent data practices help

companies demonstrate that they are following their legal obligations such as HIPAA for health data in the US, GDPR in the EU, and censoring PII. The regulatory frameworks around the world are in a constant state of flux, so the UN Trade and Development's page for data privacy laws around the world is a good resource to consult for the latest views on the global regulatory landscape.

To evaluate safety, one popular tool currently available is LM-Eval. LM-Eval allows you to test generative AI models across a wide range of task-specific benchmarks and also to build your own benchmarks to test your trained model against. This tool is compatible with Kubernetes via the TrustyAI Kubernetes Operator, and is installed by default in Red Hat OpenShift AI. Check out the TrustyAI website to learn more about using LM-Eval in Kubernetes and see how LM-Eval fits in with the rest of your cluster.

AI Guardrails

AI guardrails are a new software concept that act as safety mechanisms for LLMs. These are solutions that act as middleware between an incoming request and an LLM, and add limitations to prevent outputs that would be harmful or go

against some norm. For example, a security guardrail could detect errant PII in an output and remove it before returning it to a user. Guardrails should be able to enforce an enterprise's policies and guidelines, enable contextual understanding, and be regularly updated.

One community building an entire suite of open source AI safety tools is TrustyAI, who also created LM-Eval, mentioned in the previous section. They also provide AI guardrail tooling, such as `trustyai-detoxify`, a Python module within the larger TrustyAI Python library that provides guardrails around toxic language, among other safety tools centered on toxic language. Additionally, they maintain TrustyAI Guardrails, which acts as a server for calling detectors (such as a toxic language detector) to help developers implement their own guardrails.

LLM guardrail detectors are tools that ensure LLMs operate within predefined ethical, safety, and operational boundaries. These are designed to identify and mitigate undesirable outcomes, such as generating harmful or inappropriate content, ensuring that AI systems behave responsibly. There are a wide array of detectors available today, and you can access many public ones via the Guardrails AI Hub. Some broad examples of detectors include:

PII detectors

These screen outputs for PII and prevent it from reaching the user.

Hate, abuse, profanity (HAP) detectors

These screen outputs for toxic content, bias, or harmful content like hate speech, discrimination, or misinformation.

Bias detectors

These analyze outputs for signs of unfair biases related to race, gender, religion, or other attributes.

Prompt injection detectors

These recognize and counter potential adversarial prompts or attempts to manipulate the model into generating harmful content.

TrustyAI provides a Kubernetes Operator that allows you to seamlessly integrate it with your cluster, as well as a custom explainer for KServe.

While these toolkits are not exhaustive, they are currently available resources to help address abuses of LLMs, keeping your users and your enterprise safe. To dive deeper into

mitigating bias and harm throughout the AI development lifecycle, consult Aileen Nielsen's book *Practical Fairness* (O'Reilly, 2020).

Chapter 6. Summary and Outlook

In this report, you learned about the multilayered lifecycles that govern AI projects (the AI development lifecycle in Figure 1-1 and the AI model lifecycle in Figure 3-1) and open source Kubernetes-based tools that work to enable each of those phases at scale. You learned how to leverage open source tools to successfully move a generative AI model through these cycles, standardizing the process of model creation and allowing you to confidently deploy and manage AI models in production.

To understand how these phases interact, which open source tools to use at each phase, and how they look in practice, let's look at an example project.

Personalized Healthcare Chatbot

In this example, let's follow a fictional generative AI team at a major health insurer as it pitches and builds a personalized health chatbot. In the first phase of the AI development lifecycle, the project is initiated.

During *project initiation*, the generative AI team lead meets with the team's organization's director of engineering, head of sales, head of IT, and director of research and development to discuss a project idea floated by a member of the team. She pitches a personalized chatbot that healthcare subscribers can interact with to navigate questions about their personal health, their insurance policy, and healthcare providers. She claims this can reduce time spent by customer agents on common, personalized tasks; reduce personally identifiable information (PII) and health information from being exposed to customer agents; and increase a subscriber's agency over their own healthcare, reducing costs and increasing satisfaction.

The business units are convinced by the case she made, but the director of engineering is skeptical. How will data be safeguarded? What kinds of technologies will be used? How can we deploy this and serve all of our customers? The team lead explains this to the director of engineering, who is satisfied with her answers. We'll break down her plan throughout the rest of the chapter.

Once the project initiation is developed and agreed upon, the next phase, shown in Figure 1-1, is *data preparation*. Our technical lead's team gets to work collecting data to train and personalize the chatbot. The team chooses a popular *foundation*

model, which it will fine-tune to have better access to nonidentifiable company-wide information, and then will use techniques such as prompt engineering and retrieval-augmented generation (RAG) at inference time with a user's personal information to further personalize individual chatbot sessions. Within this phase is the first phase of the AI model lifecycle from Figure 3-1: *gathering training/fine-tuning data*. The team builds data ingestion pipelines from its healthcare partners and internal systems, *online analytics processing (OLAP)* databases to store this data, and object storage technologies to store additional data and views.

Next, the generative AI team enters the *model experimentation* phase of the AI development lifecycle. This is an iterative phase that includes the following phases of the AI model lifecycle (Fig. 3-1):

- Developing training/fine-tuning code
- Executing the training/fine-tuning job
- Evaluating the trained model

The team spends several months working on this, building the initial model training code scaffolding on which the team will fine-tune the foundation model and test RAG techniques and different prompts. The team committed to a cloud-agnostic

open source platform to allow greater flexibility across many environments, and so chose to build its infrastructure with Kubernetes. Because fine-tuning a foundation LLM is less intensive than training one from scratch, the team chose to use PyTorch libraries to fine-tune an existing smaller foundation model on a small corpus of the company's data. Early exploratory versions were created in a small Jupyter notebook environment, but as the fine-tuning datasets, base models, and fine-tuned models grew in size, the team turned to Kubeflow and the Kubeflow Training Operator to scale up the fine-tuning process on its Kubernetes cluster.

As the team trained and evaluated new versions of its fine-tuned model, managing the training clusters became a headache, and so the team decided to invest time in finding a training resource management tool. The team had become more familiar with Kubernetes at this point, and wanted to ensure it had plenty of control without too many abstractions getting in its way. The team adopted Kueue to queue up and prioritize resource-hungry training jobs, ensuring the highest-priority jobs would be run first.

One thing the team decided early on in the project, however, was that it would need an experiment tracking tool. The team knew it would be repeating the fine-tuning/evaluation cycle

frequently before the first candidate was ready to be promoted to production across many data scientists, and needed a way to understand who did what. Because the team had previously chosen Kubeflow as its platform of choice, the team was able to use Kubeflow Pipelines to build repeatable training jobs and the Kubeflow Model Registry to keep track of trained models. This allowed the data science team to keep track of fine-tuned model artifacts, prompt artifacts, and model evaluation metrics, making the team lead's life easier when deciding when to bring a model into production.

The key deliverables for this phase are production-ready model artifacts and a reusable training pipeline that accelerates both this phase and the periodic retraining of the deployed model. The pipeline itself is cleanly versioned using GitOps (see Chapter 3) principles and Argo CD to manage continuous delivery of clean production pipeline versions that will be used to train production models.

At the same time, engineers on the generative AI team are working together with the product engineering team to design and build APIs and artifact storage that allow the generative AI team to deploy new models autonomously and the product engineering team to build a chat interface that doesn't need to know any details about the model. This is the *application*

integration phase of the model development lifecycle, and for smaller teams, this may happen only after the first production-ready model is trained.

The "last" phase (in quotes because this is a cyclical, iterative process) is putting the model into *production service*. In Figure 3-1, this corresponds to *promoting the model to production for inference and monitoring the served model* (see Chapter 4).

When a model is promoted to production, a system is put in place to deploy the chosen artifact and to then serve it behind an API. While it is running in production, metrics are monitored to ensure that the model is functioning as expected and that the serving infrastructure is returning results to users in a timely manner. Our generative AI team lead chose to use KServe with the vLLM runtime. She chose KServe because of its tight integration with the Kubernetes ecosystem and active developer community. She chose the vLLM runtime because it is specifically built for serving LLMs at scale and has many features and optimizations to serve a high volume of inference requests quickly. This combination also comes with a standard API on the model that the product team can access to finalize application integration and canary deployments to gradually

roll out and test new model versions with a small number of users.

The team spent some time throughout the process to define a number of metrics to keep track of for production models. Some of these came from vLLM, others came from KServe, some were human feedback scores from customers, and still others the team built. Thanks to an integration with KServe, the team's MLOps engineers are using Prometheus to visualize and monitor the metrics of the production model and respond right away to slow performance, traffic spikes, data drift, and outages.

Using Prometheus on several occasions helped the team to catch a growing scaling issue early on, and the canary deployments provided by KServe prevented the issues from affecting more than a small number of users. Following GitOps best practices allowed the team to revert the problematic infrastructure version to a previously known good version, giving the team members time to diagnose and fix any found bugs before redeploying.

During early testing, the team found that users were able to get inappropriate answers from the chatbot and that some conversations with the chatbot were perceived as rude. One of

the machine learning engineers on the team had recently read about TrustyAI Guardrails (see Chapter 5) and began an initiative to build guardrails into KServe. With Prometheus, the team was able to monitor detections of inappropriate responses and interactions as well as the customer feedback for these instances. Using the Guardrails, the team was able to reduce negative interactions by a whopping 87%.

The results were astounding: customer agents had more time to upskill and work on serving customers who had more complex issues, customers could get personalized information in a conversational interface without having to call or wait for a live agent, and her team created a blueprint for future generative AI initiatives throughout the organization.

But the team's work was not done yet, and in fact wouldn't be done until the feature was retired or superseded by another technology. Because the model was fine-tuned on organization-wide data, it would have to be periodically fine-tuned, evaluated, and redeployed to make sure it had up-to-date information. Surprisingly, this would be a simple effort requiring only one or two data scientists less than a week to complete. This is all thanks to the generative AI team lead's forward thinking by directing the creation and use of a reusable training pipeline with GitOps, model version tracking

via a registry, data versioning, and using predictable data storage.

Future Technology Outlook

What is next for this intrepid generative AI team? We foresee three broad dimensions along which innovation in AI and AI platforms will progress over the coming months:

- Model architectures, the capabilities they yield, and the tools and techniques used to create them
- The level of integration between tools underpinning the overall MLOps lifecycle and the ability to leverage these integrated solutions to build intelligent applications more cost effectively
- Further innovation in inference optimization to reduce response latency, in areas such as quantization techniques, LoRA adapters, dynamic batching, and inference workload scheduling techniques
- The ability to build and maintain AI-enabled applications in a way that ensures the responsible and trustworthy use of AI

Along the first dimension, we will continue to see the largest models getting larger as compute resources become more

performant, more efficient, and more readily available. At the same time, we will continue to see novel approaches for customizing smaller models with an organization's own data in order to yield high performance results at lower costs for domain-specific use cases. General purpose AI will become more powerful and use case-specific models will get easier to create. The key to taking advantage of these innovations will be to leverage frameworks and training platforms with strong open source community adoption in order to be well-positioned at the forefront of new technological leaps.

Along the second dimension, platform suites such as Kubeflow that bundle tools across the AI model and development lifecycles will make it easier for data scientists to use each component in a way that is increasingly transparent to them. For example, libraries for training models will natively integrate with experiment tracking and model registry tools so that data scientists' experiments are automatically tracked. Additionally, these solutions will come with tools to automatically detect and respond to hardware failures during model training and serving, reducing the overall cost of developing and running models. These projects will gain and improve capabilities for managing the cost of developing models via better management of compute resources and sharing these resources across data science teams.

Managing compute resources is the major theme of the third dimension. Here, we will see continued optimization of inference workloads via ongoing research on new quantization techniques (in which the weights and activations of a model are represented with lower precision data types, reducing memory usage), efficient utilization of key-value (KV) caches, LoRA adapters, and dynamic batching techniques (where requests to the hardware accelerator are batched based on batch size or time elapsed), all in order to make more efficient use of hardware accelerators like GPUs. We will also begin to see wholly new innovations in how inference workloads are scheduled and executed on hardware accelerators, again to use existing accelerators more efficiently.

And along the final dimension, we expect to see more resources (financial, talent, etc.), research, and tooling dedicated to the ethical and safe training and use of generative AI. From training data lineage and tracking to model explainability and safety tools like guardrails and hallucination detection, generative AI has broadly expanded the potential for harmful creation and use of LLMs. Community-driven initiatives like TrustyAI and specifically TrustyAI Guardrails, along with Guardrails Hub, are already making it easier than ever to protect users, PII, and enterprises from the various direct and indirect harms (such as lawsuits) that can be brought about

with LLMs. Additionally, we expect more norms and tooling for the ethical collection and sharing of large datasets to protect privacy and intellectual property rights.

About the Authors

Alex Corvin is a senior engineering manager responsible for crafting and executing capabilities for data scientist experimentation and model training within Red Hat OpenShift AI, Red Hat's flagship AI/ML platform. Alex has orchestrated creation and enhancement of functionalities for distributed training and fine-tuning of AI models, encompassing extensive language models, utilizing tools such as Ray and PyTorch. Alex and his team contribute heavily to several prominent open source projects including Kubeflow Pipelines, Kueue, Kuberay, Feast, and Kubeflow Training Operator. Alex has spoken at numerous industry conferences like Ray Summit, DevConf, NVIDIA GTC, OpenShift Commons, and more.

Taneem Ibrahim is a senior engineering manager whose team is responsible for several projects in the development of an enterprise-class MLOps product, Red Hat OpenShift AI. As part of the product engineering work, Taneem and his team participate in several open source projects such as model serving (KServe, ModelMesh, vLLM), responsible AI (TrustyAI, AIX360), and model registry (KubeFlow, ML Metadata). Taneem has also worked with an extensive AI partner ecosystem for integration with OpenShift AI and IBM watsonx.ai. Taneem has

spoken at many industry events like Ray Summit, Red Hat Summit, and KubeCon.

Kyle Stratis is a software engineer with over a decade of experience across the AI development lifecycle in a variety of domains, including computer vision, health technology, and social media analytics. Along with being an O'Reilly author, he is the founder of Stratis Data Labs, an AI and data consultancy, and was most recently the lead machine learning engineer at Vizit Labs, where he built Vizit's internal AI platform.