

Real-world ML systems with Kubernetes

Copyright © 2025 by Re Alvarez Parmar and Elamaran Shanmugam. All rights reserved.

Some Rights Reserved. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

To view a copy of this license,
visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

For the self-taught.

Table of Contents

Accelerating machine learning innovation using MLOps and Kubernetes.....	11
1.1 The need for speed.....	12
The Iteration Problem.....	12
What MLOps Engineering Solves.....	13
DevOps Principles, Applied to ML.....	13
Speed Is the Competitive Advantage	14
1.2 Developing machine learning systems	14
1.2.1 Establishing business goals.....	16
1.2.2 Data engineering	16
1.2.3 Model development and training	19
1.2.4 Evaluation	20
1.2.5 Inference	20
1.2.6 Observability.....	21
1.2.7 Rinse and repeat.....	22
1.3 MLOps platform.....	22
For data engineers.....	23
For machine learning engineers.....	23
1.4 Benefits of building MLOps on Kubernetes.....	24
1.5 When to choose Kubernetes as your ML platform.....	25
1.6 What does this book teach?	26
1.7 What does this book not teach?	27
1.8 Summary.....	28
Fundamentals of Kubernetes	29
2.1 Kubernetes architecture	30
2.1.1 Kubernetes control plane	30
Managed Kubernetes in the Cloud	30
API Server	31
Controller Manager	31
Scheduler	31
Cloud controller manager	31
etcd	32
2.1.2 Kubernetes data plane	32
Nodes.....	33
autoscaling nodes.....	33
CNI	34
CoreDNS	34
2.2 Kubernetes Objects.....	36
2.2.1 What are Containers?.....	36
2.2.2 Pods.....	37
Resource allocation	42
Probes	43

init containers	44
Sidecar containers.....	44
2.2.3 Deployments	45
Scale a Deployment.....	46
Horizontal Pod Autoscaling	47
Update a Deployment.....	47
Rollback a Deployment.....	48
2.2.4 Services.....	49
Labels and Selectors	51
Create a service.....	52
2.2.5 Namespaces.....	54
dns names and namespaces	55
2.2.6 Ingress.....	55
2.2.7 Adding storage to Kubernetes workloads	58
Adding storage to workloads.....	59
Attaching persistent storage to Pods.....	61
2.2.8 ConfigMaps.....	63
2.2.9 Secrets	65
2.3 Jobs	65
Cronjobs	66
2.3.1 StatefulSets.....	66
2.3.2 DaemonSets	68
2.4 Kubernetes Package Management.....	69
2.4.1 Helm.....	70
Summary.....	70
Building an ML platform in Kubernetes	72
3.1 Creating a Kubernetes cluster.....	74
3.1.1 Creating an EKS cluster using Terraform.....	74
3.2 Architecting an MLOps system on Kubernetes.....	75
3.3 Setting up an identity provider	78
3.3.1 Configuring DNS	79
3.3.2 Getting a TLS certificate.....	81
3.3.3 Deploying an Ingress Controller	82
3.3.4 Creating an Identity Provider using Keycloak	86
3.3.5 Preparing Keycloak for client authentication.....	88
3.3.6 Create a user in Keycloak	93
3.3.7 Understanding the authentication workflow	93
3.4 Creating a self-service development environment	94
3.4.1 Deploy JupyterHub	95
3.4.2 Role-Based Access Control with Keycloak	96
3.4.3 Providing persistence to notebook servers.....	97
3.4.4 Customizing user environment	99
3.4.5 Enabling GPU workloads in Kubernetes	101

3.4.6 Reducing wastage by shutting down idle notebooks	104
3.4.7 Optimizing GPU node utilization	106
3.5 Summary.....	107
Scaling Data Pipelines with Kubernetes.....	108
4.1 Understanding the basics of data processing.....	109
4.2 Data processing in pipelines	113
4.3 Introducing Apache Airflow	114
4.4 Installing Airflow	116
4.4.1 Configuring Keycloak to authenticate Airflow users	117
4.4.2 Deploying Airflow using Helm	118
4.4.3 Exploring Airflow Architecture	119
Scheduler.....	121
Workers	121
Webserver.....	122
4.4.4 Executors.....	122
4.4.5 Airflow Operators	123
4.5 Creating your first DAG	123
4.6 Loading DAGs in Airflow	126
4.7 Running your first DAG	128
4.8 Passing data between tasks	129
4.8.1 Passing large amounts of data between tasks	131
4.9 Transitioning from notebook to pipeline	133
4.9.1 Providing secrets to tasks.....	134
4.9.2 Handling dependencies	135
4.9.3 Adding data processing step to the pipeline	138
4.10 Configuring the default properties of worker Pods	139
4.11 Managing resources.....	140
4.11.1 Airflow Pools.....	141
4.11.2 Limiting resources in Kubernetes.....	142
4.12 Using Apache Spark in pipelines	143
4.12.1 Running Spark jobs on Kubernetes	144
4.12.2 Creating Airflow DAGs for Spark Job	148
4.13 Architecting for scale	150
CoreDNS	151
Picking the right executor.....	151
Scaling Airflow components	152
Summary	152
Training machine learning models on Kubernetes.....	154
5.1 Distributed training in a nutshell	156
5.2 Distributed training with TensorFlow	160
5.2.1 TensorFlow cluster.....	161
5.2.2 MultiWorkerMirroredStrategy	161
5.2.3 ParameterServerStrategy.....	162

5.3	Training models with Kubeflow	162
5.3.1	Installing Kubeflow Trainer.....	164
5.3.2	Training TensorFlow models with Kubeflow	164
5.3.3	Recapping the process.....	169
5.4	Distributed training with PyTorch	169
5.4.1	PyTorch training with Kubeflow	171
5.5	Running MPI jobs with Kubeflow.....	175
5.5.1	Fault tolerant training with Elastic Horovod.....	177
5.6	Improving efficiency with Alternate Kubernetes schedulers	178
5.7	Optimizing distributed training.....	183
	Place worker nodes as closely as possible.....	183
	Use hardware acceleration.....	184
	Tuning NCCL to Fully Utilize Cross-Host Networks.....	184
	Gradient Aggregation in float16.....	185
	Overlapping Backward Path Computation with Gradient Aggregation ...	185
5.8	Cleanup.....	185
	Summary.....	186
	Distributed computing with Ray and Kubernetes	187
6.1	Introduction to Ray	187
6.1.1	Anatomy of a Ray cluster.....	189
6.1.2	Setting up a Ray cluster with KubeRay	192
6.2	Running workloads on a Ray cluster.....	193
6.2.1	Using Ray interactively from notebooks	194
6.3	Customizing a cluster using RayCluster.....	197
6.4	Running jobs in a Ray Cluster	201
6.4.1	Submitting Ray Jobs using API	204
6.5	Adding fault tolerance to a Ray cluster	205
6.6	Running batch workloads with RayJob	206
6.7	Hyperparameter tuning with Ray.....	208
6.7.1	Tracking experiments with MLflow	210
6.8	Inference with Ray Serve.....	213
6.9	Scaling Ray Serve.....	218
6.10	Comparing Ray with Kubeflow	219
	Summary.....	221
	Operationalizing ML models with Kubernetes	222
7.1	Packaging a machine learning model	223
7.2	Why pick Kubernetes for serving models?.....	227
	Autoscaling.....	227
	Traffic distribution.....	228
	Rollout strategies	229
	Efficient resource utilization	230
	Observability	230
7.3	Containerizing a serve application.....	231

7.3.1	Storing trained models	234
7.3.2	Handling dependencies	235
7.3.3	Storing inference code	236
7.4	Best practices for serving models with Kubernetes.....	237
7.4.1	Implement high availability	238
7.4.2	Configure autoscaling	238
7.4.3	Withstand hardware failures	239
7.4.4	Prevent disruption during system maintenance	241
7.4.5	Continuous health assessment.....	242
7.5	Model Version and Lifecycle Management	244
7.6	GPU sharing	246
	Summary	247
	Serving LLMs on Kubernetes	248
8.1	Unique challenges of serving LLMs	249
8.1.1	LLM Inference Servers	250
8.2	Introducing vLLM	251
8.2.1	vLLM Support for Kubernetes.....	253
8.2.2	vLLM Production Stack Architecture.....	254
vLLM: router	254	
vLLM: Serving Engines	256	
Metrics.....	257	
8.3	Getting started with vLLM	258
8.3.1	Creating an LLM service on Kubernetes	260
8.3.2	Understanding the deployment	262
8.3.3	Hosting multiple models	264
8.3.4	Serving model variants with Multi-LoRA	267
8.3.5	Performing Updates without Downtime	269
8.3.6	Optimizing Model Loading and Startup	270
8.3.7	Loading models from object storage.....	271
8.3.8	Using MinIO as a Model Cache.....	272
8.3.9	Reducing cold startup with image caching	274
	Including Container Images in the worker node's image.....	274
	Pre-Pulling with Kubernetes Image Puller	274
	Enabling Lazy Image Loading	275
8.4	Autoscaling vLLM	275
8.4.1	Benchmarking a vLLM Deployment	278
8.5	Distributed inference with vLLM	279
8.5.1	Multi node inference	280
8.6	Embracing the ecosystem	282
8.7	Cleanup.....	282
	Summary	283
	Observing ML Applications in Kubernetes.....	285
9.1	Understanding Observability	286

9.1.1	New challenges with observing containers.....	286
9.1.2	Observability Standards in Kubernetes.....	287
9.1.3	Three Layers to Monitor	287
Observing Kubernetes Control Plane - Logs.....	288	
Observing Kubernetes Control Plane - Metrics.....	288	
Observing Kubernetes Data Plane.....	289	
Observing Kubernetes Applications.....	290	
9.2	Observing with OpenTelemetry	291
9.3	Collecting Logs with OpenTelemetry	293
9.4	Introducing Prometheus	295
9.4.1	Prometheus Exposition Formats	296
9.4.2	Choosing a monitoring system for your workloads	297
9.4.3	Installing Prometheus in Your Cluster.....	298
9.4.4	Breaking Down the kube-prometheus-stack Components.....	300
Grafana (kube-prom-stack-grafana)	300	
Prometheus Operator (kube-prometheus-operator).....	300	
kube-state-metrics (kube-state-metrics)	300	
Node Exporter (prometheus-node-exporter)	300	
Prometheus Server (kube-prome-prometheus-0).....	301	
Alertmanager	301	
9.5	Target discovery in Prometheus	301
9.5.1	Understanding Service and Pod Monitors	302
9.6	Scaling based on custom metrics.....	304
9.7	Monitoring and scaling an application	305
9.7.1	Creating a Service Monitor.....	306
9.7.2	Exposing metrics using Prometheus Adapter	308
9.7.3	Scaling workloads horizontally.....	309
9.8	Visualizing metrics with Grafana	312
9.8.1	Exploring Grafana.....	313
9.8.2	Applying these patterns across your MLOps stack.....	314
9.9	Emitting and Collecting Traces	314
9.9.1	Implementing Tracing in ML Workflows.....	316
9.9.2	Collecting Traces	317
9.10	Handling the dilemma of what to monitor	317
9.10.1	Monitoring Distributed ML Workloads.....	318
9.10.2	Monitoring GPU Utilization	319
9.10.3	Monitoring the Model.....	319
Validation Job Pattern	319	
Request Mirroring Pattern	320	
Inline Monitoring Pattern	320	
Log-Based Analysis Pattern	320	
Summary.....	321	
Creating the base infrastructure using Terraform.....	324	

main.tf explained	325
Creating VPC for an EKS cluster.....	329
Creating an EKS cluster	331
Provisioning storage for workloads.....	333
Creating a Shared Storage using Amazon EFS	335
Infrastructure costs	337
Deploying the template.....	338
Acknowledgements	340
About the authors	342

1

Accelerating machine learning innovation using MLOps and Kubernetes

This chapter covers:

- Bringing DevOps to machine learning development using MLOps
- Machine learning development lifecycle
- The anatomy of an MLOps platform
- The role of Kubernetes in MLOps

Machine learning is no longer a research project—it's infrastructure. It's operations. It's pipelines, training, and inference. From recommending what to watch next to flagging fraudulent transactions, ML systems have become core components of modern software. But while building a model in a notebook is straightforward, scaling and deploying that model reliably in production is where most teams fail. According to Gartner (<https://www.gartner.com/en/newsroom/press-releases/2018-02-13-gartner-says-nearly-half-of-cios-are-planning-to-deploy-artificial-intelligence>), up to 85% of ML projects never reach their intended impact. Why? Because the hardest part isn't the model, it's everything around it.

Operationalizing machine learning means taking a promising prototype and integrating it into a system that can scale, recover from failure, adapt to new data, and stay performant over time. This transition demands much more than

good models; it requires disciplined engineering across infrastructure, deployment, monitoring, and data pipelines.

This is where MLOps comes in. *MLOps* is the intersection of machine learning, software engineering, and DevOps, a set of practices and tools that help teams build, deploy, and maintain ML systems at scale. And today, the most versatile platform for building MLOps infrastructure is Kubernetes.

Kubernetes, the open-source container orchestration platform, has become the backbone of cloud-native software delivery. *Cloud-native* refers to designing systems that fully leverage the scalability, elasticity, and automation of the cloud. Cloud-native applications are built as loosely coupled services, deployed in containers, and managed through declarative infrastructure. Kubernetes embodies this philosophy. It's portable, extensible, and backed by a massive ecosystem of open-source tools.

For organizations already running on Kubernetes, using it to run ML workloads is a natural progression. But even for teams just getting started, Kubernetes offers a consistent and future-proof foundation for deploying machine learning systems. It allows teams to standardize deployment, scale inference workloads, and automate operations, all using the same platform trusted by modern software engineering teams to run web services, APIs, and now, AI workloads.

This book is written for data engineers, software engineers, MLOps practitioners, DevOps engineers, and platform teams who are responsible for taking models from notebooks to production and keeping them running once they're there.

This book teaches you how to design and operate scalable ML systems using Kubernetes. You'll learn the tools, patterns, and architectural principles needed to move machine learning beyond experimentation and into production safely, reliably, and repeatedly. Whether you're building an internal platform or scaling an AI product, Kubernetes gives you the foundation to operationalize machine learning like software. Our goal is to show you how you can use it to turn your ideas into real-world products.

1.1 **The need for speed**

Nowadays, machine learning isn't held back by lack of ideas. It's held back by slow feedback loops. The faster teams can run experiments, the faster they can learn, and the sooner they can ship accurate models that actually work in production. That's the core reason MLOps engineering exists.

THE ITERATION PROBLEM

Machine learning development is fundamentally iterative. You try one architecture, change a feature, tweak a hyperparameter. Most experiments don't

work. Some do, barely. You inch forward with every run. That's not a flaw in the process, it is the process, for now anyway.

But iteration is expensive. Training takes time. Environments drift. Pipelines break. Shared resources create bottlenecks. And if you're running this all manually, every new experiment becomes a burden.

Imagine you're making soup without a recipe. No clue what works, just a pantry full of ingredients and a clock ticking down. Garlic? Too much. Peanut butter? Somehow worse. Eventually, you land on something that's not awful.

Now imagine your boss walks in and says, "Great. Now do it 500 more times. By tomorrow."

That's machine learning without MLOps.

WHAT MLOPS ENGINEERING SOLVES

MLOps engineering exists to remove friction from that process. It brings discipline and automation to a space that's otherwise chaotic and fragile. The goal isn't to constrain experimentation but accelerate it.

Instead of manually launching jobs, teams define pipelines. Instead of sharing notebooks over Slack, they version code and data. Instead of guessing what broke, they check dashboards and logs. MLOps engineering provides the systems, standards, and infrastructure to make that all possible, so ML teams can move faster with more confidence.

Key outcomes of MLOps engineering include:

- Automated pipelines for data processing, model training, and deployment
- Reproducible environments using containers and configuration-as-code
- Consistent execution across cloud, on-prem, or hybrid platforms
- Observability to track model performance, drift, and failure modes
- Rapid rollback and model versioning to reduce production risk

DEVOPS PRINCIPLES, APPLIED TO ML

At its core, MLOps borrows heavily from DevOps. The practices are familiar: Continuous Integration, Continuous Delivery, infrastructure as code, automated testing. But applying them to ML requires adaptation.

ML systems aren't just code, they include large datasets, evolving labels, GPU-heavy training jobs, and models that don't behave like binaries. MLOps engineering adapts DevOps techniques to this complexity, enabling:

- Fast, safe experimentation
- Safe deployment of evolving models
- Continuous retraining and monitoring
- Isolation between users and workflows
- Infrastructure reuse across teams and use cases

Without these systems, ML teams are stuck rebuilding the same scripts, debugging the same issues, and manually managing jobs that should have been automated months ago.

SPEED IS THE COMPETITIVE ADVANTAGE

Why does this matter? Because accuracy comes from iteration, and iteration comes from speed. The faster you can test ideas, the sooner you'll find what works. This is why the most mature ML teams prioritize MLOps early. They know that shipping better models isn't about trying harder, it's about learning faster.

MLOps engineering enables short feedback loops, high-confidence deployments, and scalable experimentation. It's not overhead, it's what makes modern ML possible at scale.

This book is about building that kind of system.

1.2 *Developing machine learning systems*

Like traditional software development, developing machine learning systems requires a series of steps to guide the development and deployment of machine learning projects. Since these projects have a heavy reliance on data analytics, they benefit from following the well-established approach proposed in the Cross-Industry Standard Process for Data Mining (CRISP-DM) model.

CRISP-DM is an open standard process model outlining common approaches used by data mining experts. It is the most widely used analytics model in data mining. According to CRISP-DM, there are six major phases of data mining:

1. Business understanding
2. Data understanding
3. Data preparation
4. Model training
5. Evaluation
6. Deployment
7. Observability

Building upon CRISP-DM, the CRISP-ML (Q) framework adds monitoring and maintenance phases to address challenges that are unique to machine learning. This cyclical process ensures organizations derive practical business value from machine learning systems by following a structured approach from project inception to completion. Most machine learning projects have the following phases:

- Establishing business goals
- Data engineering
- Model development and training
- Evaluation
- Inference
- Observability

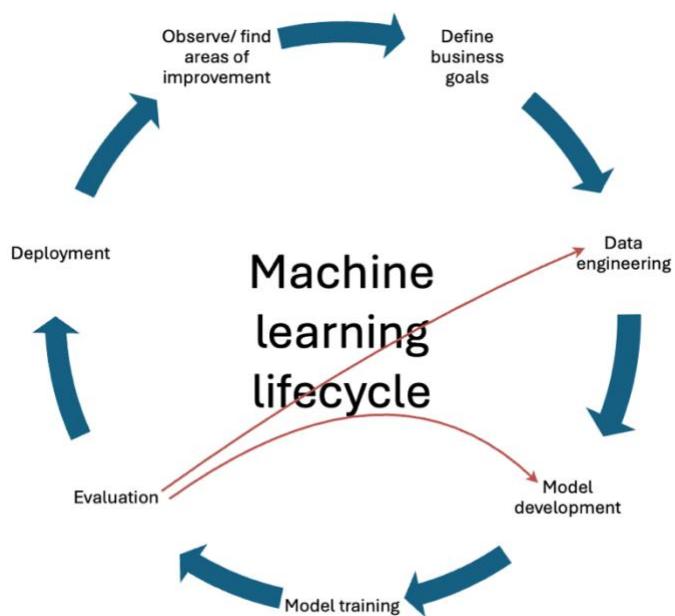


Figure 1.1 Machine learning lifecycle is a cyclical process built upon the principles of continuous improvement. The process begins with the identification of a business problem, which determines the data you'll need to create a model. Once you build and deploy a model, you monitor its performance and identify improvement opportunities for the next version.

Please don't let the graphic in figure 1.1 give you the impression that machine learning product development lifecycle is a cyclical in the strictest manner. The process often gets short-circuited. For instance, if a model you trained isn't producing desired results, you may want to go back a step or two to tweak the input dataset or the model's architecture.

Let's take a brief look at the stages of machine learning lifecycle. Later in the book, we will create these processes and get hands-on experience.

1.2.1 Establishing business goals

The success of a machine learning system hinges on having a clear understanding of the problem you aim to solve and the business objectives you want to achieve. Financial institutions, healthcare companies, and countless other institutions leverage machine learning for a myriad of use cases, from credit scoring and predictive diagnostics to fraud detection. However, the ultimate goal remains the same: to harness machine learning's potential to increase productivity and improve the organization's bottom line.

Defining specific, measurable, achievable, relevant, and time-bound (SMART) business objectives is crucial. These objectives provide serve as a guiding light, providing clear direction and purpose for the project, ensuring alignment with the organization's overall goals and strategies. They dictate data requirements, facilitate the evaluation of the system's efficacy, and ensure that machine learning initiatives directly contribute to achieving strategic outcomes.

By establishing clear business objectives, organizations can ensure that their machine learning initiatives are in line with the broader strategic goals of the organization. This alignment maximizes the impact of ML on business performance. Well-defined objectives also enable organizations to track progress, measure success, and evaluate the effectiveness of the ML system in meeting predefined goals. Outlining business objectives helps in prioritizing resources and allocating them effectively, focusing on projects that directly contribute to achieving strategic outcomes.

1.2.2 Data engineering

Data is the foundation for effective machine learning systems, just as high-quality, prepared food is essential for human thriving. The quality and quantity of data directly affect the efficacy of machine learning models - garbage input data will lead to poor, unusable outputs, even for the most sophisticated algorithms.

Building successful machine learning systems requires a large volume of high-quality data, which involves significant effort in collecting, pre-processing, cleansing, labeling, and implementing data governance. It is rare to find an off-

the-shelf dataset that perfectly meets the needs of a project, so data preparation work is crucial to address common issues like missing records, outliers, inconsistencies, and improper formatting. Effective data preparation is paramount, as most machine learning algorithms require data to be structured in a specific way to produce accurate, reliable, and actionable insights.

Data engineering is an umbrella term for a series of processes to build datasets:

- Ingestion – Collect raw data from various sources
- Exploring – Understand what's in the collected dataset. How valid is the data? Is it current? What's its uniformity? Consistency? How good is its quality?
- Cleansing – Remove errors, handle missing values, and correct issues identified during exploration.
- Structuring - Organize the raw data into a more structured and analysis-friendly format.
- Labeling – Annotate objects or entities in the dataset.
- Enriching - Add context or new information to the dataset to make it more valuable for analysis. It includes selecting, manipulating, and transforming the raw data into data formats more suitable for model training. In machine learning, this process is sometimes referred to as *feature engineering*.
- Validating - Check the data for correctness, completeness, and compliance with standards.
- Publishing - Share the cleaned and enriched data in a format that makes it most useful for the intended purpose.

All these steps are run as part of a data pipeline. A *data pipeline* contains the code that is run on raw data to produce a refined data store. If you are dealing with a system that produces data constantly, then you schedule the data pipeline to run on regular intervals. A typical pipeline

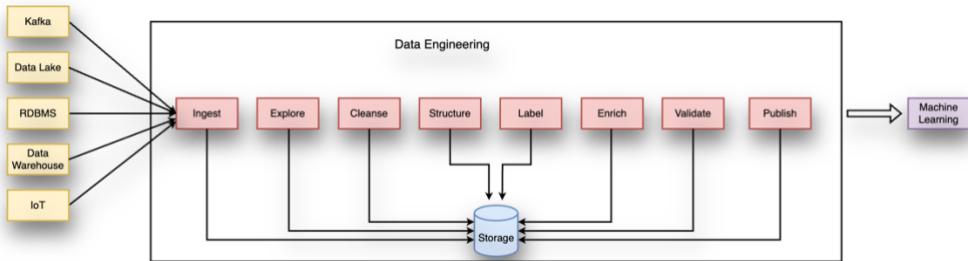


Figure 1.2 Data engineering is collecting data from various sources and preparing it to be used as the input for building machine learning models. Data engineering is executing a series of steps to improve data quality.

A workflow management system like Apache Airflow (discussed later in the book) lets you build data pipelines and schedule them. These systems run your pipelines as scheduled tasks. They retry on failure, avoid duplicate runs, and provide monitoring capabilities. One benefit of these data processing tools that is worth mentioning is that they automate many tasks for you. For example, they can let you download a file from a server. They can run a SQL query for you. Instead of writing the code to do these tasks, you can use the tool's built-in methods to perform common tasks. The idea is to reduce the cognitive load for developers.

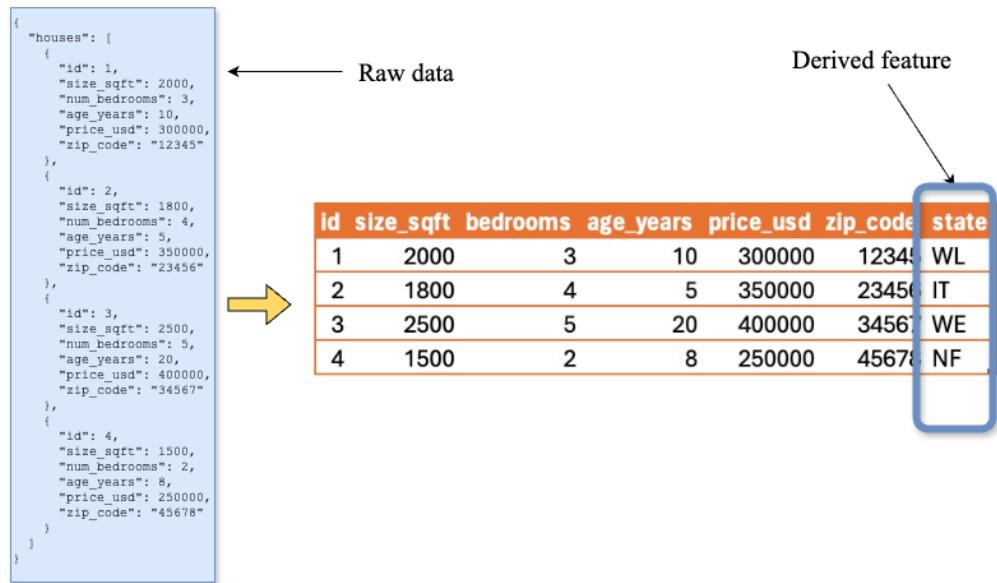
Nearly all popular workflow management tools support Kubernetes today. Whether you end up deploying a tool mentioned in this book or an alternative, you'll find Kubernetes a strong contender for data engineering. By the end of this book, you'll have a thorough understanding of the role Kubernetes plays in building a scalable data analytics platform. Not just its benefits like cost optimization, autoscaling, and hardware abstraction, but also the complexity involved.

What is a feature?

Imagine you have a dataset that contains information about houses, such as the size of the house (in square feet), the number of bedrooms, the age of the house, its price, and zip code. Each of these pieces of information – the size, number of bedrooms, age, and price – is a feature of the dataset.

Feature engineering includes converting raw data into usable features.

It is also possible to create new features deriving from other features from input data. For example, you can use zip code in the housing dataset to derive a new feature called "state".



Feature engineering is the process of converting raw data into structured, useable data. During this phase the data can be augmented by creating new features using derivation.

1.2.3 Model development and training

Once datasets are ready to be consumed, the next phase is writing the code to build an algorithm. During this phase machine learning engineers create, refine, and train models.

Developing a model may be the most iterative process of the entire machine learning development lifecycle. It requires a lot of experimentation with algorithms, data, and hyperparameters to prove hypotheses. Feature engineering continues in this phase. During model development, a machine learning engineer may learn that need additional features, in which case, they must go back to the data prep step and modify features. A preferred approach is to track every experiment and iteration systematically, recording the code, parameters, results, and data.

Infrastructure management is key challenge with training models, especially at scale. Most machine learning models require GPUs or accelerators. While it is possible to train models on CPU, larger models require GPUs to be performant.

Consider an organization with a team with five machine learning engineers actively experimenting with model code. To support their parallel work and ensure timely progress, dedicated GPU resources become essential. Since GPUs are very expensive resources, effective management of the infrastructure used for training becomes critical in controlling costs. In later chapters, you'll learn how Kubernetes solves the challenges of resource allocation and maximizing efficiency while keeping costs down.

Another aspect where Kubernetes shines is distributed training, which involves spreading model training across multiple GPUs and machines. It is beneficial for larger models and computationally demanding tasks like deep learning. This approach allows for parallel processing of data and model training tasks, which enhances the speed and efficiency of the training process.

1.2.4 Evaluation

During the development lifecycle of a machine learning project, teams iterate through numerous versions of a model, refining its accuracy and optimizing its performance until achieving a stage where the model's outputs align with desired business outcomes.

Model evaluation is an aspect of machine learning development that involves systematically recording and managing the progress of models throughout their lifecycle. It encompasses logging model parameters, code versions, metrics, and output files for every experiment.

Experiment tracking tools provide the mechanism to create *experiment runs*, which represent individual executions of model code, enabling the storage of metadata and artifacts related to each run, such as model weights and performance metrics. Tracking gives you an insight into how changes in input data and hyperparameters impact model performance and simplify comparing experiment results.

1.2.5 Inference

Once you've finished training a model, it's time to deploy it and use it. Model serving or inference is deploying and making machine learning models available for use in test and production environment. There are two ways of deploying a model. An online deployment is when a trained model is made accessible via a REST or gRPC API. In an offline deployment, the model is typically used to process large batches of data periodically, rather than responding to real-time requests.

In the training phase, a model learns patterns in the training data to make predictions. In inference, the model receives input data, which should be distinct from training data, to produce predictions without further adjustments of its parameters.

In online deployments, when a single instance of model can't keep up with the incoming requests, multiple replicas are deployed. Every replica handles a portion of the traffic, which makes it possible to send many requests to a trained model without running into hardware constraints. With Kubernetes, you can operationalize ML models and scale them to maintain high levels of reliability.

1.2.6 Observability

Distributed systems, especially those leveraging cloud-native architectures, microservices, and containers, are inherently complex and dynamic. Over the last few years, development teams have implemented observability to help manage the complexity by providing insights into the system's behavior and performance using logs, metrics, and traces.

In ML systems, observability is equally critical to ensure the health, performance, and reliability of data processes and models. Observability in ML systems have two key pillars: data observability and ML observability.

Data observability refers to the ability to fully understand, monitor, and diagnose the health and performance of data processes within a system. It provides transparency into real-time aspects of the data pipeline, including data quality, resource usage, operational metrics, system interdependencies, data lineage, and the overall health of the data infrastructure. The core ideas of data observability are:

- Data freshness – Ensuring that the most recent data is available and used.
- Data lineage – Tracing data's journey to identify data sources.
- Data volume monitoring – Keeping track of the amount of data flowing through the systems.
- Schema changes – Auditing changes to data to prevent downstream errors.
- Data quality checks – Ensuring that the data meets the quality and anomalies are caught early in the cycle.

Implementing observability in data engineering helps enhance data quality, makes troubleshooting easier, prevents downstream issues, and simplifies data auditing.

ML observability involves closely monitoring and understanding how ML models perform once they are deployed in production environments. It

encompasses monitoring the performance of the models, the entire ML pipeline, and infrastructure to ensure models are functioning as expected. It includes:

- Model performance monitoring – Tracking metrics such as accuracy, latency, and resource utilization.
- Data pipeline monitoring – Ensuring that the input data remains consistent with and representative of the training data.
- Logging and tracing – Capturing logs and traces of model predictions and system events.
- Explainability – Understanding a model's behavior by tracing its decision-making process.

ML models are trained on a specific corpus of data. Models must be retrained when the underlying dataset is no longer a true representation of the ground truth data. This means once deployed, you must monitor your model's metrics closely to know when you must update the model using fresher data. Of course, this implies that you have a robust logging, monitoring, and alerting system.

Making observability easy is one of the things Kubernetes does well. It does so by enforcing standardization. While Kubernetes doesn't come with a built-in logging or monitoring solution, community observability tools such as Prometheus and Grafana give you an end-to-end solution for monitoring your workloads. If your organization uses a SaaS-based observability solution and like to use their tools to monitor Kubernetes workloads, you can ship this observability data to them as well. There are hardly any commercial monitoring platforms that don't support Kubernetes. The real challenge is understanding what to monitor. Once you are past that question, the next question you might have to answer is how long you should keep this data for. We've dedicated an entire chapter to observability. That chapter covers popular open source tools in the observability space and discusses the things you need to know about monitoring ML workloads.

1.2.7 Rinse and repeat

Data is constantly changing in modern systems. This means that by the time you finish deploying the model, you are ready to restart the pipeline and run the process all over again. Thankfully, there's automation to give you a few moments of peace.

1.3 MLOps platform

As we have learned, building a machine learning system demands a diverse skill set. Data engineers play a crucial role in aggregating, cleansing, and managing datasets, while machine learning engineers focus on constructing models and

integrating machine learning functionalities into both new and existing systems. In some organizations the data engineer is also a machine learning engineer. While in others these responsibilities are handled by distinct teams. Regardless of the organizational structure and engineering responsibilities, success in this endeavor hinges on having a platform that offers end-to-end capabilities to tackle the challenges encountered at each stage of the machine learning system development lifecycle. This comprehensive approach ensures seamless collaboration among different roles, streamlines workflows, improves security, and facilitates the efficient deployment of machine learning solutions across various applications and environments.

An effective MLOps platform provides self-service capabilities to its users. The platform automates hardware resource provisioning and software configuration for notebooks, data processing, model training and deployment, and observability. An ideal platform seamlessly integrates into the user's workflow, allowing them to accomplish tasks with the least bit of friction.

An MLOps platform built on Kubernetes offers tools that automate and streamline repetitive tasks into workflows, the platform simplifies the development and deployment of machine learning systems. The platform delivers essential services tailored to each group involved in the development lifecycle. This may include features such as single sign-on for authentication, a unified solution for logging and monitoring, resource management tools, version control systems, collaboration platforms, and integration with various ML frameworks and libraries.

FOR DATA ENGINEERS

An MLOps platform provides a reliable infrastructure that scales to the needs of data engineering. By streamlining infrastructure management, the platform liberates data engineers from the complexities of infrastructure provisioning, allowing them to focus solely on their core task of coding for data analytics. The platform offers data analytics and workflow management tools such as Jupyter Notebooks, Spark clusters, and Airflow. These tools help data engineers analyze data interactively and create batch jobs for recurring data processing tasks.

FOR MACHINE LEARNING ENGINEERS

Machine learning engineers are the primary beneficiaries of an MLOps platform as it furnishes them with all the necessary infrastructure to streamline model development lifecycle. From providing managed Jupyter notebooks to GPUs for training and serving models, the platform facilitates tools and infrastructure needed to develop ML systems.

1.4 Benefits of building MLOps on Kubernetes

There's no one correct way of building an MLOps platform. Such a platform usually brings multiple open source and commercial tools that integrate to provide automation for machine learning scientists and data engineers. This book presents an approach to building MLOps platform using open-source tools as building blocks. Embracing open-source solutions mitigates reliance on any single cloud service provider or software vendor, ensuring flexibility and independence in platform deployment and maintenance.

An MLOps platform built on Kubernetes has the benefit of being cloud-native. It leverages the elasticity of the cloud to make workloads scalable and cost-effective. Additionally, Kubernetes offers the following advantages:

- Kubernetes has a vibrant community of open-source software builders catering to virtually all needs of software development.
- It enables the creation of immutable infrastructure, enhancing a system's consistency, predictability, and security.
- It abstracts the underlying infrastructure, freeing MLOps engineers from dealing with the intricacies of managing virtual machines, storage, and networking.
- It provides a highly scalable infrastructure for deploying and managing machine learning models. It has built-in autoscaling capabilities that help in scaling models up or down based on changing demand.
- It supports scalable distributed training to speed up model development.
- Similarly, it provides the scalable infrastructure for distributed data ingestion data analytics, providing near-native integration with tools like Airflow, Spark, and others.
- It integrates seamlessly with CI/CD pipelines, enabling automated testing, deployment, and rollout of machine learning models and applications.
- It has extensive support observability through Prometheus, Grafana, and 3rd-party tools.
- It offers capabilities to isolate workloads, making it ideal for multitenant scenarios.
- It supports policy engines can enforce security standards in clusters, ensuring compliance, security, and consistency across the environment.

Given Kubernetes' widespread success in traditional software engineering, it's unsurprising that many large organizations, including industry leaders like OpenAI (the creators of ChatGPT), opt for Kubernetes as their preferred platform for managing machine learning workloads.

1.5 When to choose Kubernetes as your ML platform

There is no shortage of open-source and commercial solutions offering platforms to machine learning teams. Many of these solutions offer end to end capabilities to builders, simplifying the entire process of building and operating ML workloads. Why should you choose Kubernetes?

This is an important question that needs rigorous deliberation. After all, Kubernetes brings its own sets of challenges to organizations. Still, Kubernetes remains the preferred platform for companies operating ML workloads at scale because:

- Many organizations already use Kubernetes to orchestrate workloads. They have sunk costs in Kubernetes. They already have Kubernetes expertise. Hence, the ML platform becomes an extension of their existing infrastructure.
- Kubernetes allows these organizations to remain cloud agnostic. They can migrate their platform to another cloud or on-premises, or they can run identical platforms across multiple clouds for business continuity and resilience.
- Kubernetes integrates seamlessly with a wide range of popular machine learning tools and frameworks. This enables you to build, test, and deploy your machine learning models in a streamlined and automated manner, leveraging Kubernetes' orchestration capabilities to manage the entire lifecycle.
- Kubernetes makes it easy to manage the underlying hardware by providing an abstraction layer.
- The distributed nature of Kubernetes allows you to distribute your ML workloads across multiple machines, speeding up the process and improving performance.
- Kubernetes offers a way to operate workloads in a reliable and fault tolerant manner.
- Kubernetes is open source and free to use.

Does that mean everyone should build their ML platform on Kubernetes? Absolutely not. Managing Kubernetes itself is a pretty onerous task, even when leveraging a managed service provider. Organizations venturing into machine learning without prior Kubernetes experience go through a steep learning curve. For this reason, using a managed cloud platform such as Amazon SageMaker or Google's Vertex AI may be a more suitable option for teams unfamiliar with Kubernetes. These platforms abstract away much of the underlying infrastructure

management, allowing data scientists and ML engineers to focus on model development and deployment.

For relatively simple or small-scale machine learning projects, the overhead and complexity of setting up and managing a Kubernetes-based MLOps platform may not be justified, and simpler solutions could be more appropriate.

Managed cloud ML platforms may provide a more streamlined and user-friendly experience compared to building a custom ML platform on Kubernetes. However, they are also less flexible and offer fewer customization options than a Kubernetes-based solution.

Choose Kubernetes when:

- Your organization is already familiar with Kubernetes.
- You prefer to build your own systems instead of buying solutions.
- You must remain cloud agnostic at all costs.

If your organization happens to be in the no-Kubernetes-experience camp, we recommend conducting a careful cost-benefit analysis before deciding to use Kubernetes for building an MLOps platform.

1.6 *What does this book teach?*

There are many ways of building an MLOps platform. Some commercial vendors, like Amazon SageMaker and Google's Vertex AI, provide a fully managed platform for MLOps. However, these platforms tie you to a specific vendor. This book will act as your guide for building a more portable MLOps platform on Kubernetes that relies on open-source software.

This book will teach you to build an MLOps platform on Kubernetes that is capable of data processing, distributed training, automated deployments, model serving, observability, and implementing security and governance.

For readers unfamiliar with Kubernetes, the second chapter covers the Kubernetes fundamentals needed to understand the rest of the book. Each subsequent chapter focuses on solving a particular problem in machine learning, such as data processing, model training, or deployment strategies. In chapter 3, you'll learn how to create a scalable development environment for data and ML engineering. Chapter 4 dives into building reliable data pipelines. Chapter 5 builds upon the previous chapter and discusses strategies for creating robust ML pipelines. Chapter 6 delves into scaling model training with distributed frameworks. In chapter 7, we create an experiment to show how to use retrieval-augmented generation (RAG). Chapter 8 focuses on deploying ML models to production and handling various inference techniques. In chapter 9, you'll learn how Kubernetes helps you incorporate observability in ML systems. Chapter 10

is dedicated to implementing security and governance in ML systems. The last chapter shows how to run generative AI (GenAI) application on Kubernetes.

As you build this platform, you'll learn about open-source tools, frameworks, and libraries that can be used as the building blocks for the system. The book provides a comprehensive guide for building a machine learning platform that can be deployed in production, using Amazon Web Services as the underlying infrastructure. Since the platform is built on Kubernetes, you can build a similar system on Microsoft Azure or Google Cloud.

To demonstrate how to build an ML platform on Kubernetes, this book will guide you through the implementation of some of the most widely adopted tools and solutions. In cases where there is no clear market leader, the book will present alternative options and selection criteria to help you determine the path best suited for your specific environment and requirements.

Rather than solely focusing on the tools themselves, the book emphasizes the underlying utility and functionality they provide. This approach allows you to understand the core capabilities needed to construct a robust MLOps platform, rather than being constrained by the limitations of any particular tool or technology.

By taking this broader, utility-centric perspective, you'll gain a deeper understanding of the essential components required to operationalize machine learning at scale. This knowledge will enable you to make informed decisions about the most appropriate tools and frameworks to incorporate into your MLOps platform, tailored to the unique needs and constraints of your organization.

The book walks you through the process of setting up a sample machine learning project and then transitioning it into a fully operational system. The book serves as a resource for individuals seeking to hone skills and knowledge to develop and operationalize machine learning systems at scale. There is an accompanying GitHub repository (<https://github.com/mlops-on-kubernetes/Book>) for this book that contains the code needed for the project.

1.7 What does this book not teach?

MLOps engineers are DevOps engineers for machine learning workloads. While some machine learning and data science knowledge is beneficial for people that identify themselves as purely MLOps engineers, their primary focus is building and maintaining the infrastructure, tools, and processes necessary to support machine learning workflows at scale.

This book does not teach writing the code for data science or machine learning algorithms. We recommend Francois Chollet's *Deep Learning with Python, Second Edition* (Manning, 2021) if you're interested in learning machine learning

development. For data science, we recommend Leonard Apeltsin's *Data Science Bookcamp* (Manning, 2021)

1.8 Summary

- Building machine learning systems is a complex hurdle that many teams fail to cross. As per Google Cloud, a production-grade ML system usually contains 5% or less ML code. The rest of the system is responsible of managing data and ML models.
- MLOps addresses challenges in machine learning development by providing a platform for machine learning scientists and data engineers.
- Developing a machine learning system is a multi-step process that starts with defining business objectives and data collection.
- Data engineering involves ingesting, cleansing, and enriching data. Feature engineering is the continuation of data engineering during model development.
- Model development is an iterative process that requires several rounds of iteration before getting desired results.
- Kubernetes provides a scalable bedrock for building cloud agnostic MLOps platforms.
- This book presents an MLOps architecture built on Kubernetes and other open-source tools, helping you avoid the pitfalls in developing large scale machine learning systems.

2

Fundamentals of Kubernetes

This chapter covers:

- An overview of Kubernetes architecture
- Exploring the key components and Kubernetes resources
- Deploying applications in a Kubernetes cluster
- Packaging application for distribution

Kubernetes (pronounced Koo-buhr-nay-tees) is a platform for managing distributed systems. Its first version was based on an internal project at Google called Borg. Kubernetes implemented Borg's distributed design principles in Go language. Since 2015, a community of developers has managed Kubernetes under Cloud Native Computing Foundation (CNCF). At its core, Kubernetes is a highly extensible orchestration system for containerized workloads.

Kubernetes is a container orchestration platform that has gained significant popularity in recent years. One of the key reasons why many organizations have adopted Kubernetes is its ability to be *cloud-agnostic*. This means that Kubernetes can be deployed and run on various cloud providers, such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), among others. By using Kubernetes, organizations can avoid being locked into a specific cloud provider and have the flexibility to migrate their applications between different clouds or even run them on their own on-premises infrastructure. This cloud-agnostic nature of Kubernetes enables organizations to choose the cloud provider that best meets their requirements in terms of cost, performance, and availability, while also future-proofing their infrastructure investments. Additionally, Kubernetes provides a consistent and unified management interface

across different clouds, simplifying the management and deployment of applications and infrastructure in a multi-cloud or hybrid cloud environment.

In this chapter, you'll learn the fundamentals of Kubernetes. As one can imagine, a system that offers end to end workload management capabilities must be complex. It is an impossible task to teach you about all things Kubernetes within a chapter. Unless that chapter is six-hundred pages long. This chapter summarizes the components that you must know to operate machine learning systems in Kubernetes. We recommend Marko Lukša and Kevin Conner's *Kubernetes in Action, Second Edition* (Manning, 2020) or Josh Rosso, Rich Lander, Alex Brand, and John Harris's *Production Kubernetes* (O'Reilly, 2021) for readers that want to build an in-depth understanding of Kubernetes. The next sections will get you started so you have a firm grasp on concepts that you will use to build upon in later chapters.

In this book, Kubernetes acts as the meta-platform on which you'll deploy open-source tools to build a robust ML platform. These tools will provide various functionalities such a platform needs. As you learned in the previous chapter, the ML code is a fractional part of a scalable ML system. Besides the ML code, such a system requires infrastructure automation, pipelines, and supporting services such as user authentication, observability, model management, and experiment tracking. By using Kubernetes as the common platform, you can ensure a consistent and reproducible environment for your ML workflows, from development to production.

2.1 Kubernetes architecture

A Kubernetes cluster has two major components: control plane and data plane. The *control plane* comprises the internal components that Kubernetes needs to function. The *data plane* is where your applications run.

2.1.1 Kubernetes control plane

The brains of a Kubernetes cluster live in the control plane. It comprises a highly available set of servers running Kubernetes' system processes and a database. On the left side of Figure 2.1, you can see the depiction of the control plane components.

MANAGED KUBERNETES IN THE CLOUD

When you use a managed Kubernetes service provided by a cloud vendor, such as Amazon EKS, Google GKE, or Azure AKS, the control plane is managed by the cloud provider. This means that the cloud provider is responsible for maintaining, scaling, and upgrading the control plane. They handle tasks like monitoring, troubleshooting, and performing necessary maintenance on the control plane.

This allows you to focus your efforts on deploying and managing your application workloads on the Kubernetes data plane, which is the part of the cluster where your containers and pods run. Let's look at what's behind a Kubernetes control plane.

API SERVER

Every operation in Kubernetes, whether it is user-initiated or system-initiated, is performed using APIs. The API server, known as *kube-api server*, is the central control point of the Kubernetes cluster. It exposes the Kubernetes API, which is used by both the internal and external components to interact with the cluster. The API server handles REST operations and provides the frontend to the cluster's shared state.

CONTROLLER MANAGER

The *controller manager* runs a collection of controllers that manage the state of the cluster. It ensures that the current state of the cluster matches the desired state defined by the user.

NOTE In busier clusters, the controller manager processes do rate-limit operations in a cluster. This limitation becomes more pronounced when running workloads that spawn hundreds of processes concurrently. If you want the system to go from 0 to 100, Kubernetes may do it in steps.

SCHEDULER

As the name suggests, the *scheduler* decides where to run a particular workload. When deploying an application in Kubernetes, the scheduler chooses the best-suited host to run the application.

CLOUD CONTROLLER MANAGER

In a cloud environment, Kubernetes integrates with the underlying cloud provider. The Cloud Controller Manager (CCM) manages cloud specific resources like virtual machines, networking, and storage.

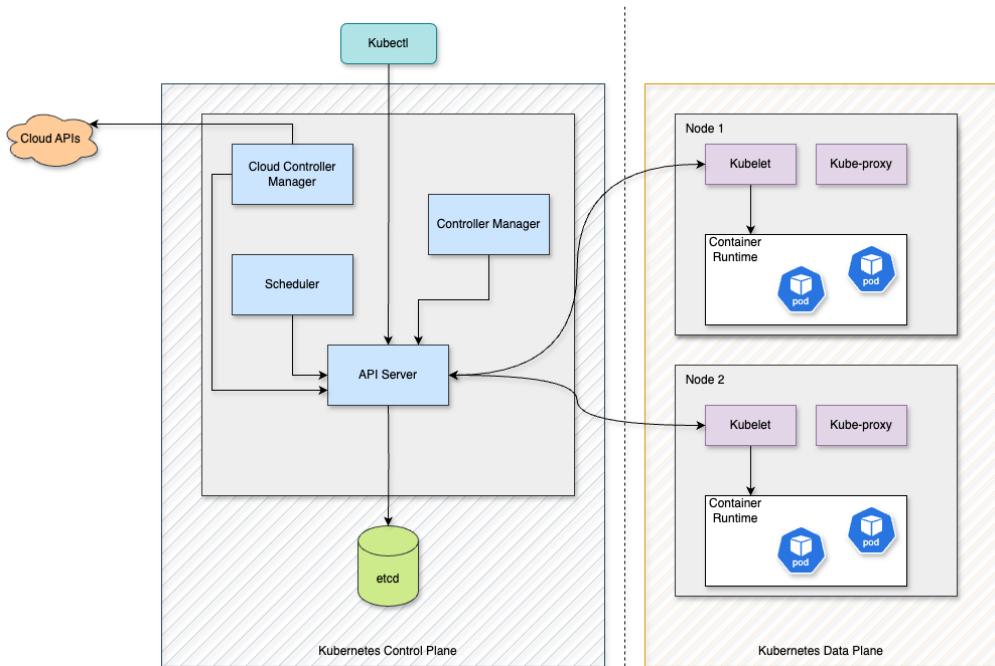


Figure 2.1 A Kubernetes cluster is made up of a control plane and data plane. The control plane runs processes that regulate the state of a cluster. User workloads run inside the data plane.

ETCD

A Kubernetes cluster maintains its state by persisting it in a highly-available etcd key-value store. The Kubernetes API server interacts with etcd to read and write this state, ensuring that the cluster's actual state matches the desired state.

2.1.2 Kubernetes data plane

In Kubernetes, the data plane consists of the infrastructure that executes the workloads you deploy. Technically, you can run workloads on the same hosts that run a cluster's control plane. This is an uncommon practice that's only available in self-managed Kubernetes clusters. Managed Kubernetes providers don't allow deploying workloads on the control plane. The key components of the data are:

- Worker nodes – Virtual or physical machines that run your containers.
- Container Network Interface (CNI) – Provides networking for containers.
- CoreDNS – DNS server in a cluster.

NODES

Hosts that run your containerized workloads are referred to as *nodes*. These can be Linux or Windows running on a virtual machine or a bare metal host. There are two types of nodes in a Kubernetes cluster: master nodes and worker nodes.

The nodes that run the Kubernetes API server and other control plane components are called *master nodes*. When you use a managed Kubernetes service (like Amazon EKS), you don't have access to the master nodes. You cannot SSH into them or check their health. The cloud provider manages these servers for you.

When you deploy a workload in your cluster, its containers are usually not run on the master nodes. This segregation ensures that the control plane has dedicated resources. User workloads are run on a different set of machines known as *worker nodes*. While master nodes run Kubernetes control plane processes, worker nodes run user-defined containers.

Every node runs three processes:

- Container runtime – responsible for creating, running, stopping containers. Examples are containerd, docker, CRI-O.
- The kubelet – acts as the agent between the Kubernetes control plane and the container runtime.
- Kube-proxy – implements Kubernetes network abstraction by maintaining network (typically iptables) rules.
- Container Network Interface – provides networking for containers.

AUTOSCALING NODES

A typical Kubernetes data plane is made up of a group of worker nodes that provide resources to run workloads. A cluster may start with just a couple of worker nodes. As you deploy more and more applications to the cluster, that resource utilization on that solitary node increases. Over time, the node becomes saturated, running out of resources to run more workloads. At this stage, your cluster runs out of capacity. When this happens, you must add more nodes to your cluster to accommodate additional workloads.

TIP The standard way of automating the scale in and out of nodes is by using Kubernetes Cluster Autoscaler, which is an open-source project maintained

by the Kubernetes community. If you are on AWS or Azure, we recommend using Karpenster, which is also an open-source project that's designed to improve Kubernetes cluster autoscaling experience in cloud environments. It provisions and scales right-sized worker nodes based on workload demands. As a result, implementing Karpenster usually lowers data plane costs. You can read more about Karpenster at <https://karpenster.sh/docs/>.

CNI

Container Network Interface (CNI) is a framework that provides network connectivity to containers that run inside a Kubernetes cluster. Kubernetes lets you to pick a CNI based on your needs. Most cloud providers have a CNI that works with their managed Kubernetes cluster. Most users stay with the default CNI that their cluster provides. There are also open source CNIs like Cilium and Calico that offer additional features that a cloud provider CNIs don't.

COREDNS

CoreDNS is the default DNS server used in Kubernetes. It handles name resolution and provides service discovery within a cluster.

The primary way to interact with Kubernetes is using the *kubectl* command-line tool. The CLI tool allows users to create, read, update, and delete Kubernetes resources. This is one tool you'll install on your local machine. Kubectl translates your actions into Kubernetes API server calls, presenting the response in a user-friendly format.

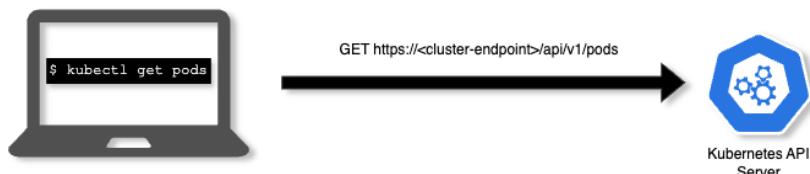


Figure 2.2 Users interface with a Kubernetes cluster using kubectl. Kubectl converts your actions into Kubernetes API server calls, presenting the response in a user-friendly format.

Kubernetes Dashboard (<https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>) is an optional component that provides a web user interface for cluster management and monitoring. The Dashboard doesn't provide all functionalities that kubectl does however.

Table 2.1 A list of commonly used kubectl commands.

Command	Operation

kubectl get	List all resources
kubectl describe	Describe a resource
kubectl delete	Delete a resource
kubectl edit	Edit a resource
kubectl logs	View container logs
kubectl patch	Update a resource
kubectl apply -f filename.yaml	Create or update a resource
kubectl config view	View kubectl configuration
kubectl events	Show events
kubectl exec	Execute a command inside a running container
kubectl port-forward	Forward a port on the local machine to a Pod
kubectl run -- image=<container_image>	Create a Pod from an image

Please see Kubernetes documentation (<https://kubernetes.io/docs/reference/kubectl/quick-reference/>) for a full list of commands and flags Kubectl provides.

Creating a Kubernetes cluster

The upcoming sections focus on managing workloads in a Kubernetes cluster. You don't need a Kubernetes cluster for this chapter. If you'd like to follow along, you can create a temporary Kubernetes cluster on your computer using tools such as Minikube. Another option is to use Killercoda (<https://killercoda.com/playgrounds/scenario/kubernetes>). Killercoda gives you access to temporary Kubernetes environments with limited features for educational purposes.

Because the system we're building in this book requires components like load balancers, virtual machines, and GPUs (optionally), once you have a fundamental understanding of Kubernetes, in the next chapter we will deploy a Kubernetes cluster in the Cloud. The goal of this chapter is to familiarize you with managing workloads in a Kubernetes cluster.

2.2 Kubernetes Objects

To run workloads in Kubernetes, you create objects. Kubernetes defines *objects* as persistent entities in the Kubernetes system that represent the state of the cluster. Kubernetes objects are "records of intent" - when you create an object, the Kubernetes system will constantly work to ensure that object exists, and its desired state is maintained. Each Kubernetes object has required fields like apiVersion, kind, metadata, and spec that define its configuration and desired state.

2.2.1 What are Containers?

Containers are bundled packages that contain all code, dependencies, and configuration needed to run an application. Popularized by Docker in 2013, containers have become the de facto standard of packaging, distributing, and deploying applications.

By encapsulating an application and its dependencies within a container image, containers make it easy to distribute software. Containers enable consistent deployment across environments, ensuring the exact build tested is deployed to production, minimizing unexpected changes and enhancing reliability. This "build once, run anywhere" approach significantly reduces environment-specific issues and streamlines the development-to-production pipeline. When used with a container orchestrator like Kubernetes, containers simplify application development, deployment, and scaling. They offer benefits such as efficient resource utilization, portability across various platforms, simplified management of application dependencies, and enhanced scalability through features like automated rollouts and self-healing mechanisms.

A container image (also known as a Docker image) is an artifact that contains an application's code, dependencies, and configuration. It is like a zip archive or a tarball. To create a container image, you create a Dockerfile, which adds your application's code to a base image. Here's a Dockerfile to create a container image for a Python application:

Listing 2.1 A Dockerfile for a Python application

```
FROM python:3.9 #A
WORKDIR /app #B
COPY . /app #C
RUN pip install --no-cache-dir -r requirements.txt #D
CMD ["python", "app.py"] #E
#A specifies the base image to use. It's using the official Python 3.9 image as a
starting point.
```

```
#B sets the working directory inside the container to /app.  
#C copies all files from the current directory (where the Dockerfile is located) into  
the /app directory in the container.  
#D runs a command to install Python dependencies:  
#E sets the default command to run when the container starts. It will  
execute python app.py
```

To build an image using this Dockerfile, you'll run `docker build . -t <name>:<tag>`. The image tag is an optional field that should never be optional in production to ensure consistency. If you don't specify a tag, it defaults to "latest". Tags are used to identify the version or variants of an image. In the Dockerfile example mentioned above, we specified that we want to use Python image with tag=3.9 as the base image.

Once you build a container image, you'll push the image to a container registry, where it is stored for easy distribution.

A container is an instantiation of a container image. You create containers from container images. For example, to run an Nginx web server using containers, you can find the specific variant of Nginx you want to run at Docker Hub, and run it like this:

```
$ docker run nginx:stable
```

This command downloads the container image from Docker Hub, creates a container from the image, and starts the container.

2.2.2 Pods

A *Pod* is the most fundamental unit of compute in Kubernetes. Kubernetes uses Linux isolation mechanisms, such as namespace and cgroups, to create and manage Pods. A Pod contains one or more containers. Typically, there is one container that runs the main application and *sidecar* containers that run processes that provide helper functions.

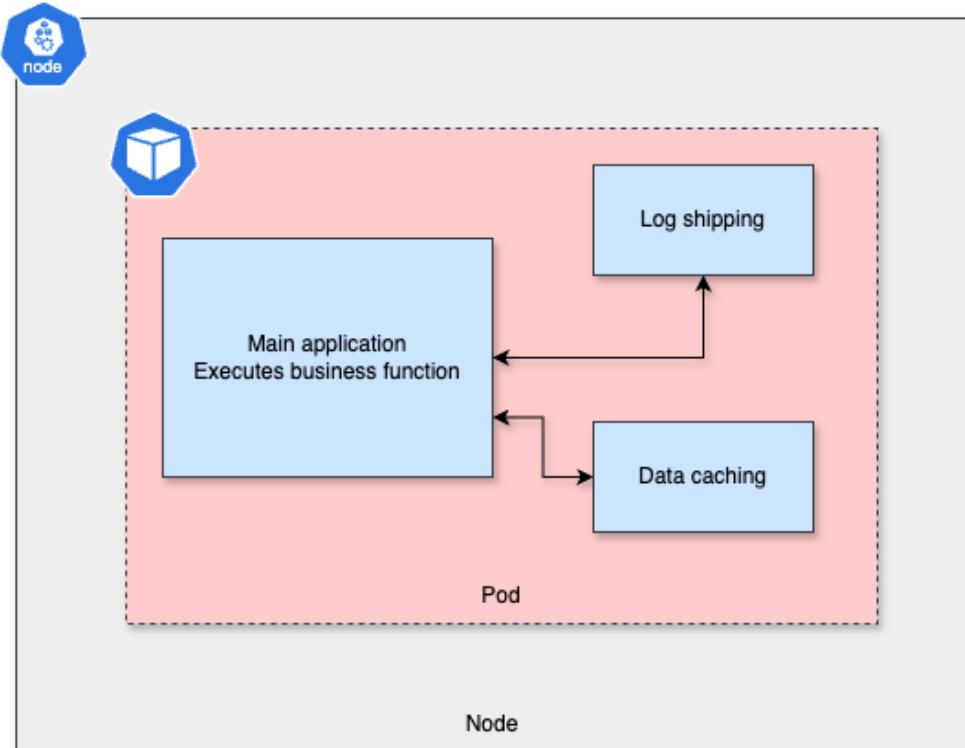


Figure 2.3 A Pod is a collection of containers that always run on the same node. Typically, a Pod has a main application container. The Pod may also run other containers that provide helper functionality to the main application.

For example, you may have a Pod that runs a Python process in the main container along with a sidecar container that handles shipping the logs the Python application. Sidecars isolate infrastructure-related code from the main application, improving maintainability by giving you the ability to update specific parts of your application.

Here's a Pod manifest with two containers:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    app: my-pod
```

```
spec:  
  containers:  
    - name: webserver  
      image: nginx:stable  
    - name: sidecar-container  
      image: quay.io/curl/curl:8.6.0  
      command: ["/bin/sleep", "infinity"]
```

To create this Pod, save the manifest as a .yaml file and run:

```
$ kubectl apply -f <filename>.yaml  
pod/my-pod created
```

View the running Pod:

```
$ kubectl get pods my-pod  
NAME      READY   STATUS    RESTARTS   AGE  
my-pod   2/2     Running   0          5m4s
```

This Pod has two containers, nginx and curl. You can see the logs the nginx container produces:

```
$ kubectl logs my-pod -c webserver
```

To stream logs from the sidecar container as they get created, run:

```
$ kubectl logs my-pod -c sidecar-container -f
```

All containers within a Pod run on the same node. They also share the Linux network namespace, which enables them to intercommunicate using localhost. Let's test network connectivity between the two containers. Let's access the nginx server running inside the nginx containers from the curl container using curl:

```
$ kubectl exec my-pod --tty --stdin --container sidecar-  
container -- curl http://localhost  
<!DOCTYPE html>  
<html>  
<head>  
<title>Welcome to nginx!</title>  
...
```

Next, run kubectl describe pods my-pod and view details about the Pod. See if you can find the IP address of the Pod in the output. An easier way to view a Pod's IP address is by running:

```
$ kubectl get pods my-pod -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NOMINATED NODE
READINESS GATES						
my-pod	2/2	Running	0	2m21s	192.168.170.70	
					<none>	

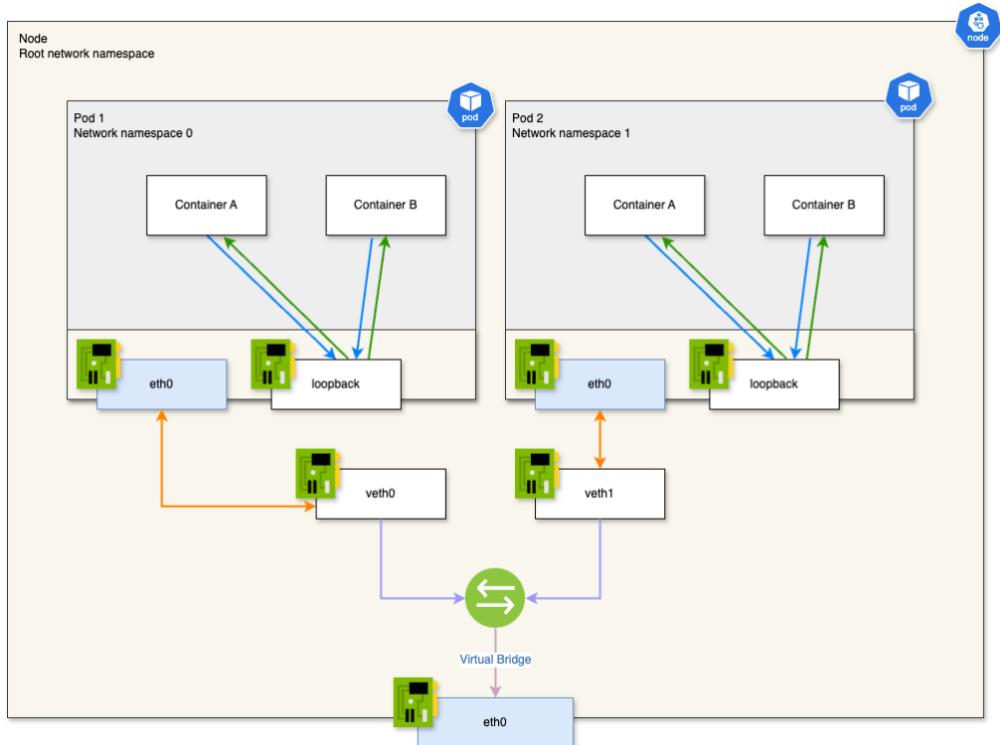


Figure 2.4 Containers in a Pod always run on the same node. They can intercommunicate using the loopback interface. Every Pod gets an IP address where it is accessible. Pods intercommunicate with each other using Pod's IP address (or DNS name).

As you can see, the Pod has an IP address. Other Pods in the cluster can connect to this Pod. But you can't connect to it just yet. That's because you have no route to the cluster's internal network.

Kubernetes allows you create temporary port forwarding, so you can test an application that is unreachable from outside the cluster. Set up port forwarding:

```
$ kubectl port-forward my-pod 8090:80
Forwarding from 127.0.0.1:8090 -> 80
```

```
Forwarding from [::1]:8090 -> 80
Handling connection for 8090
```

Open browser on your computer and navigate to `http://localhost:8090`. You should see the default web page. If you are on Killercoda, you'd have to get the address using its user interface. The result should be the good ole nginx welcome message.

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org. Commercial support is available at nginx.com.

Thank you for using nginx.

Figure 2.5 Kubernetes provides port forwarding functionality to connect to an application running inside a cluster. Port forwarding allows you to connect to an application that's only accessible from within the cluster without exposing it outside the cluster.

When you create a port forward, the traffic is tunneled through the Kubernetes API server using a single HTTP connection. Kubectl sends the traffic to the API server, which then transmits it to the Kubelet running on the node, which forwards traffic to the nginx container.

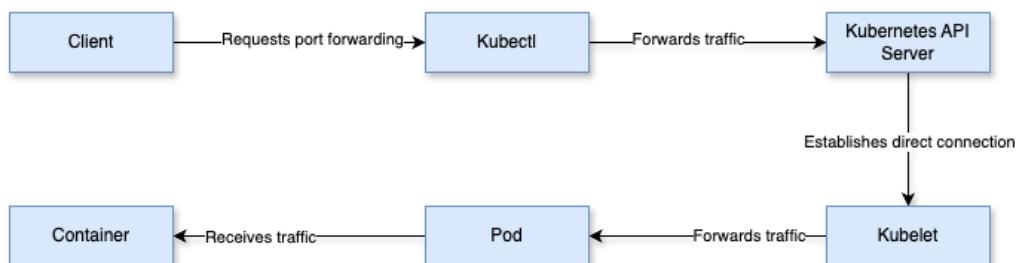


Figure 2.6 Port forwarding tunnels traffic through the Kubernetes API server and Kubelet. This tunnel is temporary and exists only for the duration of the `kubectl port-forward` command.

You can stop port-forwarding by hitting `^C` or `ctrl+C`. When you are done exploring, you can delete the Pod:

```
$ kubectl delete pods my-pod
pod "my-pod" deleted
```

RESOURCE ALLOCATION

Within a Pod, you can define how many compute resources (for example CPU, memory, and GPU) each container gets. You define the minimum and maximum resources a container gets using *requests* and *limits*, respectively. Kubernetes scheduler considers resource requirements when allocating a Pod to a node. If no nodes can meet the minimum resource requirements specified by a Pod, Kubernetes will prevent the Pod from being scheduled onto any node. As a result, the Pod will remain in a Pending state until a suitable node with adequate resources becomes available for scheduling. This ensures that Pods are only deployed on nodes that can guarantee the necessary resources, preventing potential resource contention.

If a container exceeds its resource consumption beyond the values defined in its limits, the container will either be throttled or killed, depending on the type of resource.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: nginx-container
    image: nginx
    resources:
      limits:
        cpu: "500m"
        memory: "200Mi"
      requests:
        cpu: "200m"
        memory: "100Mi"
```

The `nginx-container` in this Pod gets a minimum of 200 milli-CPU, which is equivalent to 20% of a logical CPU or a CPU core. The container can use up to 500 milli-CPU, beyond which it will be throttled. The container is also guaranteed a minimum of 100 mebibytes of memory. Since the memory limit is configured to 200 mebibytes, when its memory usage exceeds beyond the defined value, Kubernetes will terminate the Pod.

Many machine learning workloads require GPU resource. To run GPU workloads, you need a node with one or more GPUs. When a node has a GPU, you can allocate the GPU to a Pod using limits:

```
resources:  
  limits:  
    nvidia.com/gpu: 1
```

PROBES

Kubernetes can detect failure in an application container and restart or stop sending it more work. Probes are a mechanism to determine the health and readiness of containers in a Pod. Probes are defined per container. Kubernetes provides three types of probes:

- Liveness probe – Determines if a container is still running or functioning properly. This probe is used for detecting runtime issues or deadlock situations. If the liveness probe fails, Kubernetes considers the container to be unhealthy and restarts the container.
- Readiness probe – Determines if a container is ready to receive incoming traffic and serve requests. It ensures that a container has completed its initialization process and is prepared to handle requests. If this probe fails, Kubernetes will stop directing traffic to the Pod.
- Startup probe – Determines if a container's application has started successfully. It is useful in scenarios where an application takes longer time to start but can eventually become healthy. Unlike liveness and readiness probes, the result of a startup probe does not affect the container's readiness or liveness status. Once the startup probe succeeds, the liveness and readiness probes take over.

NOTE: Exercise caution when implementing probes in your application.

Improperly configured probes can become significant liabilities, potentially causing unwarranted container restarts or disrupting traffic flow to Pods.

Kubernetes users have run into some of these pitfalls and these cases are well-documented. Before configuring probes, research on misconfigured probes to avoid common errors.

There are three methods you can use to determine the health of a container:

- HTTP or gRPC Probes – Sends an HTTP or gRPC request to a specified endpoint within the container and expects a response with a specific status code.
- TCP Probe – Opens a TCP connection to a specified port within the container to check its health.

- Exec Probe – Runs a command inside the container and expects a zero exit status to indicate that the container is functioning correctly.

Here's an example of a liveness probe that uses a TCP probe:

```
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 20
  periodSeconds: 30
```

Kubernetes will probe this container 20 seconds after it starts. The health check will occur every 30 seconds.

INIT CONTAINERS

Kubernetes starts all containers in a Pod in simultaneously. There will be times when you have workloads that require some setup or preparation work before starting up. This is where init containers come in.

Init containers are special containers that run before the main application containers start. Their job is to handle any initialization or configuration tasks that need to happen before the main app can start up.

For example, let's say your app needs to connect to a database, but it needs to wait for the database to be ready before it can start. You could use an init container to check if the database is available, and only once that check passes would the main app container start.

Another common use case is downloading files or setting up directories that the main app will need. The init container can take care of that preparatory work, ensuring everything is ready to go before the main app kicks off.

If you add multiple init containers to a Pod, Kubernetes runs them in the order they appear in the Pod's spec.

SIDECAR CONTAINERS

Kubernetes 1.29 introduced support for sidecar containers using init containers. By default, Kubernetes doesn't restart an init container once it has finished running. Init containers usually perform their tasks at Pod startup and exit once they are done.

If an init container's restart policy is set to "Always", Kubernetes considers it to be a sidecar container. Sidecar containers will remain running during a Pod's lifetime. The benefit of running sidecars as init containers is that Kubernetes will start these containers before starting main containers.

```
apiVersion: v1
```

```

kind: Pod
metadata:
  name: my-pod
  labels:
    app: my-pod
spec:
  containers:
    - name: webserver
      image: nginx:stable
  initContainers:
    - name: sidecar-container
      image: quay.io/curl/curl:8.6.0
      command: ["/bin/sleep", "infinity"]
  restartPolicy: Always

```

2.2.3 Deployments

Earlier you learned how to deploy an application in a Pod. In practice, applications are deployed in a Kubernetes cluster by creating Deployments. A *Deployment* contains declarative configuration that instruct Kubernetes how you'd like to manage and update a workload.

Deployments build upon Pods, giving you the ability to manage a group of identical Pods as a single resource. A Deployment is made up of one or more Pods. When you create a Deployment, you're telling Kubernetes exactly how you want your application to look and behave. You specify things like the number of replicas you want running, the container image to use, and any other configuration details.

Listing 2.2: A sample Deployment manifest

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment #A
  labels:
    app: my-app
spec:
  replicas: 1 #B
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: webserver
          image: nginx:1.24.0-alpine #C
          ports:
            - containerPort: 80 #D

```

```
- name: sidecar-container
  image: quay.io/curl/curl:8.6.0
  command: ["/bin/sleep", "infinity"]
```

#A Name of the deployment

#B The minimum number of Pods this Deployment expects

#C Container image and tag for the Pods this Deployment creates

#D Container port to expose

Save the manifest in a `.yaml` file and create a Deployment. You'll use this deployment later in this chapter. You can view the state of your deployment by running `kubectl get deployments`.

Deployments manage the lifecycle of workloads in Kubernetes. The Deployment specification defines the desired number of replicas, a Pod template, and update strategy.

Behind the scenes, when you create a Deployment, Kubernetes creates a *Replica Set*, which represents a group of Pods. Replica Set maintains the number of desired replicas. For example, if you set the number of replicas to 5, Kubernetes will run at least 5 replicas (Pods) at all times, given that the cluster has resources to run them. If a Pod fails (or the node it's running on fails), Kubernetes will create a new Pod to maintain the desired replica count. It is beneficial to understand what Replica Sets are, but you'll almost never create a Replica Set yourself.

SCALE A DEPLOYMENT

Most modern, cloud-native workloads are well-suited for *horizontal scaling*, which involves scaling a workload by increasing or decreasing the number of identical application servers. This contrasts with *vertical scaling*, which would involve scaling by increasing the resources (CPU, memory) of a single instance.

Horizontal scaling is particularly useful for web applications and other stateless services. In these cases, you can easily adjust the number of web or application server instances based on the incoming traffic and resource demands. As traffic increases, you can scale out by adding more replicas to handle the load. Conversely, when traffic decreases, you can scale in by reducing the number of replicas, optimizing resource utilization.

Here's how you scale a Deployment using `kubectl`:

```
kubectl scale deployments/my-deployment --replicas=2
```

Kubernetes Deployments are a perfect fit for this horizontal scaling approach. The Deployment abstraction allows you to declaratively define the desired number of replicas for your application. Kubernetes will then ensure that the actual state matches the desired state by creating or deleting Pods as needed.

HORIZONTAL POD AUTOSCALING

Kubernetes allows you to scale a Deployment automatically based on CPU or memory consumption or custom metrics. *Horizontal Pod Autoscaler (HPA)* is a Kubernetes feature that automatically scales the number of Pods in a Deployment. The HPA automatically increases or decreases the number of replicas in a deployment based on the current resource usage. The HPA has configurable scaling policies that define the minimum and maximum number of Pods, as well as the rate at which Pods are added or removed.

UPDATE A DEPLOYMENT

Deployments make it easier for you rollout updates to your applications. The standard practice is to create a new container image for every application version. Once you've built a new image with a new tag and pushed it to a container registry, you can update the image field of the application's Deployment.

When you update the image, Deployment will create a new Pod. This new Pod will run a container with the newer image. When the new Pod is in ready state, Kubernetes will terminate the old Pod. If you have more than one replica in your Deployment, Kubernetes will do a rolling update until all Pods use the updated image.

Rolling updates is the default deployment strategy. Kubernetes also provides a "Recreate" strategy. This strategy is used in situations when the old and newer version of the application cannot run simultaneously. When you use this strategy, Kubernetes will terminate all old Pods before creating new ones. There will be a short period when your application will be unavailable while Kubernetes replaces Pods.

Before you update the test deployment, note that you're running version 1.24.0 of nginx. You can use kubectl to check the version of nginx:

```
$ kubectl exec deployment/my-deployment --container webserver -  
ti -- nginx -v  
nginx version: nginx/1.24.0
```

NOTE. The short options `-i` and `-t` are the same as `--stdin` and `--tty`.
`--container` or `-c` is used to specify a container in a multi-container Pod.

Update the deployment to use a newer version of nginx image:

```
$ kubectl set image deployment/my-deployment  
webserver=nginx:1.25.4-alpine-slim
```

```
deployment.apps/my-deployment image updated
```

You can watch as Kubernetes replaces the Pods by using the `-w` (shortcut of `--watch`) option with `kubectl get pods`:

```
$ kubectl get pods -w
```

The `--watch` option for `kubectl get` commands allows you to watch for changes to resources and see the updates as they happen. The `--watch` option causes `kubectl` to continuously print the current state of the selected resources and then update the output whenever the state changes. This provides a real-time view of the resource status.

Once the new Pod is running, notice that the nginx version has changed:

```
$ kubectl exec deployments/my-deployment -c webserver -ti --  
nginx -v  
nginx version: nginx/1.25.4
```

In the exercise you just performed, you updated the image using `kubectl set image`. A better practice is to save the update the image in the Deployment's manifest file and applying the changes using `kubectl apply -f <filename>`. In the real world, where deployment pipelines are automated, deployments are triggered by a continuous integration (CI) system.

ROLLBACK A DEPLOYMENT

When things go wrong with updates, as they inevitably do, Kubernetes makes it easy to rollback a deployment. You can view the history of deployments using:

```
$ kubectl rollout history deployment my-deployment  
deployment.apps/my-deployment  
REVISION  CHANGE-CAUSE  
1          <none>  
2          <none>
```

Let's say the newer version of your application has a bug and you need to undo the changes. To revert to the previous version of your deployment, execute the following command:

```
$ kubectl rollout undo deployment my-deployment  
deployment.apps/my-deployment rolled back
```

You can also provide a specific revision to rollback to a previous configuration of the Deployment resource:

```
$ kubectl rollout undo --to-revision 2 deployment my-deployment
```

```
deployment.apps/my-deployment rolled back
```

2.2.4 Services

When you created a Pod, it received an IP address, using which other apps in the cluster can connect to it. But Pods are ephemeral beings. As you scale or update your Deployment, its Pods will be replaced, and they may have different IP addresses. You need a reliable way to access your application.

In Kubernetes, Services are like a friendly concierge that helps your apps find each other and work together smoothly in your Kubernetes cluster.

Imagine you have a bunch of different apps running in your Kubernetes cluster - maybe a website, a database, and a few other services. These apps need to talk to each other, but it can be tricky to keep track of where each one is located and how to connect to them.

That's where Kubernetes Services come in. A Service acts as a stable network endpoint that your apps can use to find and communicate with each other. A Service gives you a stable IP address and DNS name that doesn't change as you scale and update Pods.

There are different types of Kubernetes Services, each serving a specific purpose:

- ClusterIP – Exposes the Service on a cluster-internal IP, allowing other applications running in the cluster to access it.
- NodePort – Exposes the Service on each node's IP at a static port, making it accessible from outside the cluster.
- LoadBalancer – Provisions a cloud-provided load balancer and assigns it a fixed, external IP to access the Service from outside the cluster.
- ExternalName – Maps the Service to an external DNS name, allowing internal applications to access external services.

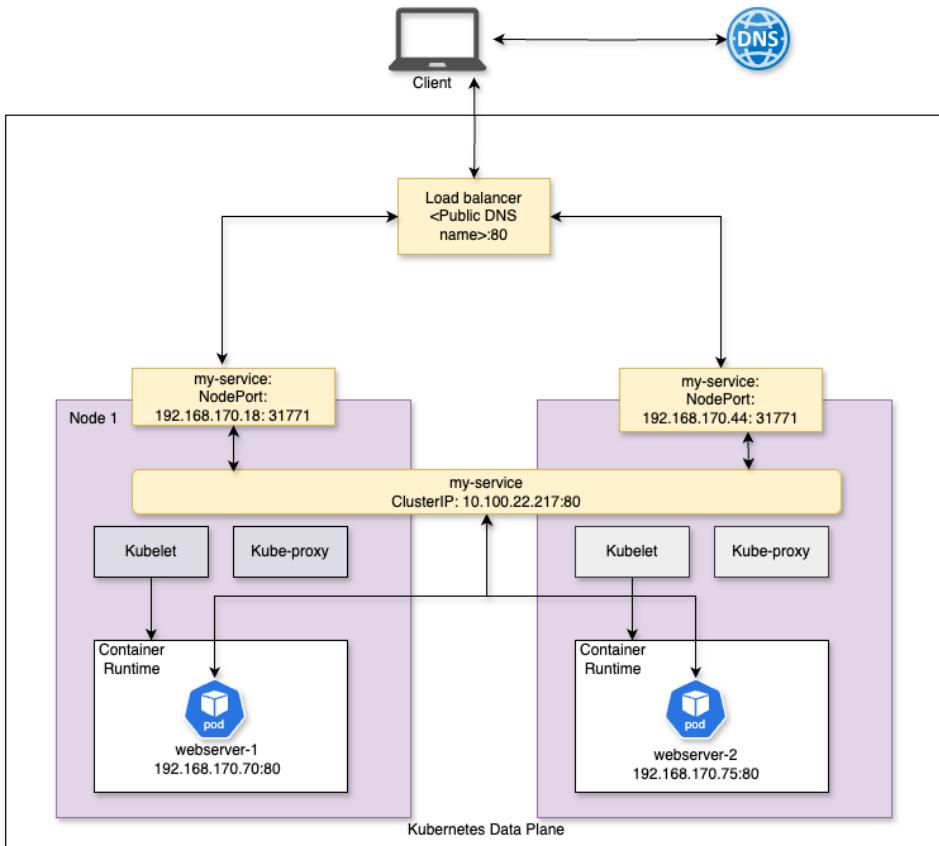


Fig 2.7 A Kubernetes Service of type LoadBalancer is designed to expose an application to the public internet by provisioning a cloud load balancer. The LoadBalancer Service will have a public IP address or DNS name that external users can connect to. The load balancer then forwards traffic to the appropriate Kubernetes Service and pods.

In the cloud, you'll mostly deal with ClusterIP and LoadBalancer Service types. ClusterIP Services are perfect for internal applications that don't need to be accessed from outside the cluster. These are services that are only used by other apps running within your cluster.

Whenever you need to expose an application outside of the Kubernetes cluster, you'll want to use a *LoadBalancer Service*. This Service type provisions a cloud-provided load balancer, like an Elastic Load Balancer on AWS or a Load Balancer on Google Cloud and provides a public-facing DNS name and IP address

for your application. This allows users and external systems to access your application from outside the cluster.

The key benefit of Kubernetes Services is that they handle a lot of the networking complexity for you. You don't have to worry about things like load balancing or service discovery - the Service takes care of it all.

LABELS AND SELECTORS

A Service uses labels and selectors to identify the group of Pods to route traffic to. So, if you have three Pods running your website, the associated Service will make sure any requests to that website get distributed across all three Pods.

The Pods in the Deployment you've created have a label attached.

```
template:  
  metadata:  
    labels:  
      app: my-app
```

Labels and Selectors are fundamental concepts in Kubernetes that enable powerful resource management and targeting capabilities. Here's how they work:

- Labels:
 - Labels are key-value pairs attached to Kubernetes objects like Pods, Deployments, Services, nodes, and others we haven't covered yet.
 - They provide identifying metadata about the objects, allowing you to organize and select subsets of resources.
 - Labels can be added, modified, or removed at any time, even after the object is created.
 - They can be system generated or user-defined.
 - Labels are used to group related objects together, such as all Pods belonging to a specific application.
- Selectors:
 - Selectors are used to target a set of objects based on their labels.
 - Kubernetes supports two types of selectors:
 - Equality-based selectors match objects based on equality of label values (e.g., app=my-app, env!=prod).
 - Set-based selectors: Match objects based on a set of label values (e.g., app in (nginx, apache), env notin (dev, test), partition).

Services use Selectors to determine which Pods to route traffic to. They are also used in Deployments to specify the Pods that should be managed.

- The relationship between Labels and Selectors is crucial:
 - Labels are applied to Kubernetes objects to provide identifying metadata.
 - Selectors are used to select a subset of those objects based on their labels.

Labels and Selectors allow you to decouple the identification of objects from the logic that targets them.

CREATE A SERVICE

Let's create a Service for the my-deployment Deployment you created earlier:

```
$ kubectl expose deployment my-deployment --port 80  
service/my-deployment exposed
```

To list the Service, run:

```
$ kubectl get service my-deployment  
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP  
PORT(S)  
my-deployment   ClusterIP  10.100.22.217  <none>  
80/TCP
```

With a Service, you now have an IP address to reach your application. In the example above, the my-deployment service is accessible at 10.100.22.217. Since you didn't specify the type of Service in the command, Kubernetes created a ClusterIP Services, as that is the default.

Any traffic sent to Service's cluster IP will be forward to Pods that match the label app=my-app, which will be all the Pods in the Deployment you've created. You can view the list of Pods that are attached to a Service by listing the Service's Endpoints.

```
$ kubectl get endpoints my-deployment  
NAME            ENDPOINTS          AGE  
my-deployment   192.168.172.131:80,192.168.177.103:80  20h
```

You can see that IP addresses of the two Pods created by your Deployment are automatically added to the Service's endpoints.

Creating a Service not only gives you a consistent IP address, but it also provides a DNS name. For example, the service you just created is accessible at <http://my-deployment>. You can validate that by running:

```
$ kubectl exec deployments/my-deployment -ti \
  -c sidecar-container -- curl http://my-deployment
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...

```

In production, you won't use `kubectl expose` to explicitly create a Service for your application. In Kubernetes, the standard practice is to create resources declaratively using manifest file, which looks like something like this for a Service resource:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: my-app
    name: my-deployment
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: my-app
  type: ClusterIP
```

Once you create a manifest, you use `kubectl apply -f <filename>` to deploy resources. You can merge multiple manifests into a single YAML file by using `---` as a separator.

Listing 2.3: Merging multiple manifests into one

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-svc
...
---
apiVersion: apps/v1
kind: Deployment
...
```

2.2.5 Namespaces

Imagine you have a big backyard that's shared by a bunch of your friends and neighbors. You all want to use the backyard, but you don't want anyone accidentally messing with each other's stuff. That's kind of like having a Kubernetes cluster that's used by multiple teams.

The teams all want to use the same Kubernetes cluster to run their apps and projects, but they need to keep everything organized and separate. That's where Kubernetes Namespaces come in handy.

Namespaces are like having different sections or "zones" in the backyard, where each team can set up their own little area. That way, the teams can make changes within their allocated namespace without worrying about someone from another team accidentally modifying something that belongs to them.

It's all about creating a sense of order and ownership. With namespaces, each team can deploy their objects, like apps and services, into their own designated area of the Kubernetes cluster. They can manage and modify their own stuff without interfering with what the other teams are doing.

Every Kubernetes cluster comes with a few built-in namespaces. You can list namespaces using kubectl:

```
$ kubectl get namespaces
NAME          STATUS   AGE
default       Active   10d
kube-node-lease Active   10d
kube-public   Active   10d
kube-system   Active   10d
```

So far, you've been creating resources in the default namespace. To see what's running in the kube-system namespace, you can specify `--namespace` (or `-n`) parameter when using kubectl. For example, to list all Pods in the kube-system namespace, run:

```
$ kubectl -n kube-system get pods
coredns-7559f9965c-gbtmm           1/1    Running
0                                     10d
coredns-7559f9965c-schmz           1/1    Running
0                                     10d
aws-node-bkktm                     2/2    Running
0                                     7d1h
aws-node-splz2                      2/2    Running
0                                     8d
kube-proxy-f9jzh                    1/1    Running
0                                     7d1h
kube-proxy-fpjkl                     1/1    Running
0                                     10d
```

You can also use `--all-namespaces` or `-A` to include resources in all namespaces. For example, you can run `kubectl get pods --all-namespaces` to get a list of all Pods running in your cluster.

DNS NAMES AND NAMESPACES

Kubernetes Services are assigned a DNS name that follows a specific format, and this format is influenced by the Kubernetes namespace in which the Service is defined. A Pod can connect to any service in the same namespace using just the name of the services. When you created a Service earlier, you could call it by using its name, which was `http://my-deployment`. However, if a Pod must call a service in another namespace, it must add the target's namespace to the DNS name.

Kubernetes assigns Services multiple DNS names. For example, the Service you've deployed is not only accessible at `http://my-deployment`, but also at:

- `http://my-deployment.default`
- `http://my-deployment.default.svc`
- `http://my-deployment.default.svc.cluster.local`

The "default" part is because you've created the service in the default namespace. The DNS names of Service follow this pattern:

- `<service-name>`
- `<service-name>.<namespace>`
- `<service-name>.<namespace>.svc`
- `<service-name>.<namespace>.cluster.local`

When a Pod connects to a Service in another namespace, it will have to use one of the DNS names that includes the Service's namespace.

2.2.6 Ingress

You have learned that to make a service available outside the cluster, you create a Service of type LoadBalancer. A LoadBalancer Service creates an entry point for a specific Service. In many cases, you don't need one entry point per Service, but a single load balancer that fronts many Services.

Consider a case of a Kubernetes cluster that's running a bunch of different microservices, each providing a specific piece of functionality for your overall system. For example, you might have a "frontend" service that's accessible at `example.com/site`, and an "api" service that's accessible at `example.com/api`.

If you were to create a LoadBalancer Service for each of your microservices (frontend and API), each service gets its own public-facing IP address and DNS name users could access directly (site.example.com and api.example.com). The advantage here is simplicity - each service has its own dedicated load balancer, which can be convenient for certain use cases. The downside is that you'd need to manage multiple load balancers, which can get unwieldy and costly as the number of services grows.

Alternatively, you could use a single Ingress resource to handle routing for both the "frontend" and "API" services. In this setup, the Ingress acts as a central traffic cop, looking at the URL path or hostname that users are accessing (site.example.com or api.example.com) and then forwarding the traffic to the appropriate Kubernetes Service. This allows you to consolidate all your external access points into a single Ingress, which can be more efficient and easier to manage as your cluster grows. The Ingress can also provide additional features like SSL/TLS termination, URL rewriting, and more advanced routing rules.

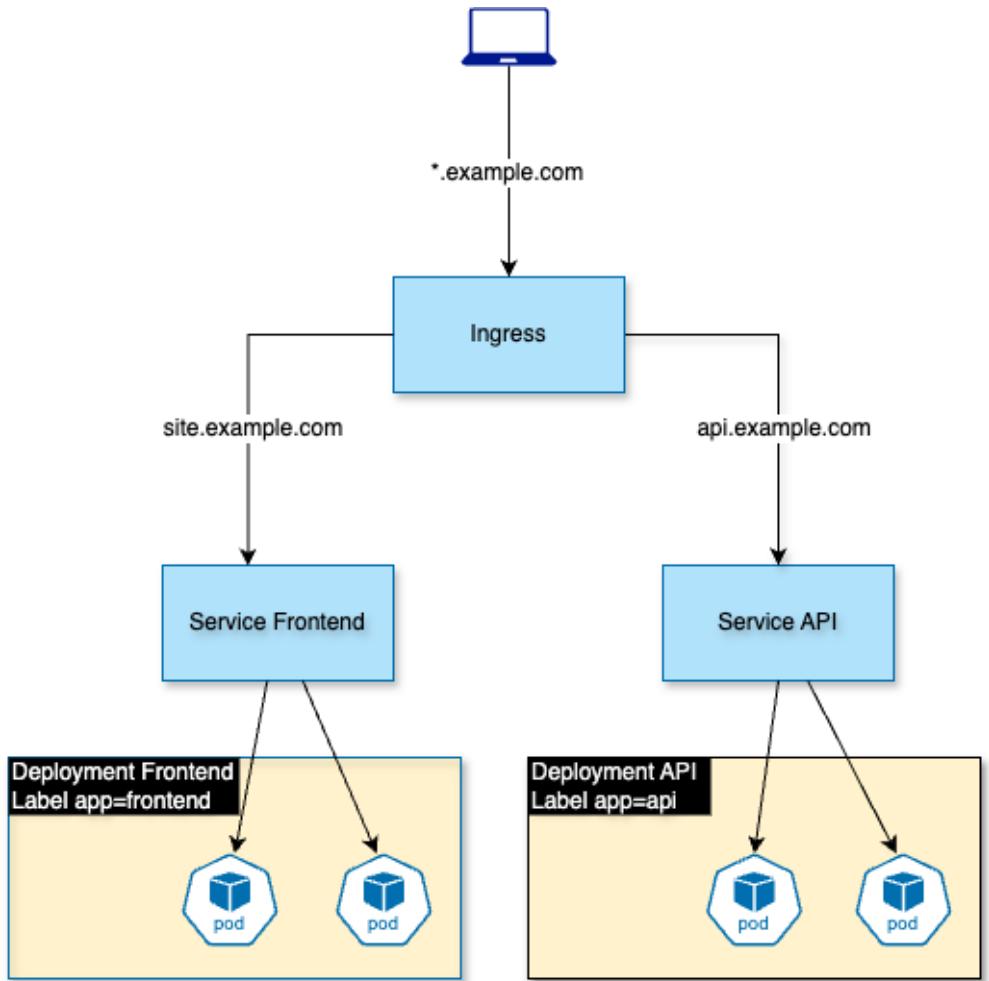


Figure 2.8 An ingress routes traffic to Services based on host and path of the URL. They allow you to use a single Load Balancer to front many backends.

Ingresses operate at the application layer, also known as Layer 7 of the OSI model. This means that Ingresses can make routing decisions based on higher-level, application-specific information, such as the URL path, hostname, or even the content of the HTTP request. They can perform tasks like:

- URL-based routing – Directing traffic based on the URL path or hostname
- TLS/SSL termination – Handling the encryption and decryption of HTTPS

traffic.

- Name-based virtual hosting – Serving multiple hostnames (e.g., example.com and example.org) from a single IP address.
- Intelligent load balancing – Distributing traffic across multiple backend Pods based on advanced algorithms.
- Middleware and transformations - Applying additional processing, such as authentication, rate limiting, or request/response modification.

The key difference between LoadBalancer Services and Ingress is that Ingresses give you more advanced routing capabilities, while LoadBalancer Services provide a simpler, more direct way to expose a single service externally.

2.2.7 Adding storage to Kubernetes workloads

Containers are designed to be ephemeral - meaning that any new data or changes made within the container's filesystem during runtime are not persistent. When a container stops or gets terminated, all the data stored inside that container is wiped out and lost forever. For applications that persist data on the local filesystem, Kubernetes introduces the concept of Volumes.

Volumes are a way to attach persistent storage to a container, allowing the application running inside the container to read and write data to a designated storage location. Kubernetes Volumes can be:

- emptyDir – A temporary directory that exists during the lifetime of a Pod. This is useful for storing transient data or to exchange data between containers. Data stored in this volume type doesn't get deleted if a container restarts. Data can reside on the node's disk or memory.
- hostPath – Mounts a directory from the host node's filesystem into the container. This is useful for development but not recommended for production use.
- Local – Mounts a directory, partition, or disk from a specific node.
- NFS – Mounts an NFS share. This is used when many Pods need access to shared data.
- PersistentVolumeClaim – Creates and mounts a volume from a storage provider such as AWS EBS, GCP Persistent Disk, or a local storage volume.

The data used for building machine learning systems is typically stored in data lakes or databases, often leveraging object storage services like Amazon S3 or Google Cloud Storage. These object storage services provide low-cost, scalable, and durable storage for the large volumes of data required by modern ML workloads.

Many data engineering libraries and frameworks, such as Spark, Pandas, and TensorFlow, provide standardized methods for storing and retrieving data from these object storage systems. This allows data scientists and ML engineers to focus on building and training their models, without having to worry about the underlying storage infrastructure. Cloud object storage services like Amazon S3 are a great fit for data-hungry machine learning workloads, offering handy features like versioning, lifecycle management, and the ability to replicate data across different regions.

A key reason for the popularity of object storage systems is that they play nicely with the most popular ML frameworks and libraries. Data scientists and engineers can easily integrate their pipeline code to read from and write to these object stores, without having to worry about the nitty-gritty details.

Now, while object storage has become a natural fit for machine learning, most other applications running on Kubernetes don't persist their data directly in these cloud-based object stores. Instead, they read and write data to the filesystem, which is generally backed by a network-connected storage service, or appliance in on-premises environments.

This makes sense, especially in the cloud where compute and data services are decoupled. Kubernetes provides a friendly way to attach storage for your applications by provisioning and managing storage for these more traditional applications. The apps don't even need to know the specifics of the underlying storage system - they just read and write to the local filesystem.

To make this storage integration as smooth as possible, the Kubernetes community has created Container Storage Interface (CSI). CSI is like a universal translator that allows storage vendors to develop special plugins, or "drivers," for their products. This way, Kubernetes can easily communicate with many storage systems, whether it's block storage, file storage, or even object storage.

The best part about the CSI approach is that it keeps the storage implementation separate from the Kubernetes core. Storage vendors can update and improve their CSI drivers independently, without having to wait for new Kubernetes releases. It benefits both Kubernetes users and storage providers - users get access to the newest storage features, and providers can innovate at their own speed.

ADDING STORAGE TO WORKLOADS

To add a network volume to a Pod, you can either create a volume yourself or let Kubernetes create it on your behalf.

Kubernetes offers two modes for provisioning storage: static and dynamic.

In *static provisioning*, Kubernetes administrators manually allocate and configure storage volumes (such as an Amazon EBS volume) before they are

needed by Pods. Once storage has been provisioned, it is attached to a Pod by creating Persistent Volume (PV) and Persistent Volume Claim (PVC) resources. Behind the scenes, the CSI driver of the storage system provisions backend resources and makes these resources available to Pods.

In Kubernetes, PVCs are requests for storage made by users (for example applications running in pods). A PVC specifies the source (NFS, Amazon EBS, local) and the properties of storage (access modes, size, class) needed by the application.

Listing 2.4: A Persistent Volume Claim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-claim
spec:
  storageClassName: "ebs"
  volumeName: my-pv
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
```

When you create a PVC, Kubernetes tries to find a suitable PV that can satisfy the PVC's requirements. If a match is found, the PV and PVC are bound together, and the application can use the storage provided by the PV. If there is no suitable PV available, Kubernetes can dynamically provision a new PV based on a Storage Class (SC). The Storage Class defines the parameters for the dynamically provisioned PV.

Listing 2.5: A sample storage class

```
apiVersion: storage.k8s.io/v1
kind: StorageClass #A
metadata:
  name: ebs #B
provisioner: ebs.csi.aws.com #C
volumeBindingMode: WaitForFirstConsumer #D
#A Specifies that we're creating a StorageClass resource using the storage.k8s.io/v1 API.
#B Name of the StorageClass is set to "ebs". This name can be referenced when creating PVCs.
#C The volume plugin (CSI driver) used to provision PVs.
#D Delays the binding and provisioning of a PV until a Pod using the PVC is created.
```

PVs are the actual storage resources, such as an NFS share, an EBS volume, or a Persistent Disk. Manually creating storage resources can be ideal for environments with predictable storage requirements but is complex to manage

and maintain at scale. Many storage providers support dynamic provisioning of persistent volumes, which automates the provisioning process and reduces manual effort. This provisioning mode is well-suited for dynamic workloads and cloud-native environments. Instead of pre-provisioning storage, you can let Kubernetes manage the lifecycle of volumes.

Depending on the storage service you pick, you'll either create a PV manually (static provisioning) or let Kubernetes create one for you using dynamic provisioning.

Listing 2.6 A statically provisioned PV

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv #A
spec:
  accessModes:
    - ReadWriteOnce #B
  capacity:
    storage: 20Gi #C
  csi:
    driver: ebs.csi.aws.com #D
#A Name of the PV
#B PV's access mode
#C Size of the volume
#D The CSI driver for the storage system (in this case EBS).
```

A PV can have three possible access modes:

- **ReadWriteOnce** – The PV can only be attached to one Pod or Node at a time.
- **ReadOnlyMany** – The PV can be attached to multiple Pods but in read only mode.
- **ReadWriteMany** – The PV can be attached to multiple Pods for read and write operations.

Besides usage, the access mode also depends on the storage system. For example, you cannot attach an Amazon EBS volume to multiple Pods. As a result, you can only use **ReadWriteOnce** access mode with EBS. Shared storage services like Amazon EFS or Amazon FSx for Lustre are used when multiple Pods need a shared filesystem.

ATTACHING PERSISTENT STORAGE TO PODS

Pods can request a storage volume by specifying a PVC. Kubernetes will then find a suitable PV and attach it to the Pod, or a new PV will be created if dynamic provisioning is supported by the CSI driver.

Listing 2.7 A Pod with dynamically allocated persistent storage

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pv-pod
spec:
  containers:
    - name: nginx-container
      image: nginx
      volumeMounts:
        - name: persistent-storage
          mountPath: /data
  volumes:
    - name: persistent-storage
      persistentVolumeClaim:
        claimName: ebs-claim
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-claim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: ebs-sc
  resources:
    requests:
      storage: 4Gi
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-sc
provisioner: ebs.csi.aws.com
volumeBindingMode: WaitForFirstConsumer
```

The Pod that Listing 2.7 creates gets a 4 Gi EBS volume attached. The volume is mounted at /data path inside the container. Kubernetes will use the PVC as a template to create the PV for the Pod. The volume binding mode defined in the storage class tells Kubernetes to create the PV only once a Pod needs one. View the mounted volume by listing block devices inside the container:

```
$ kubectl exec my-pv-pod -ti -- lsblk
NAME      MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
nvme0n1    259:0   0 100G  0 disk
| -nvme0n1p1  259:1   0 100G  0 part /etc/resolv.conf
|           |               /etc/hostname
|           |               /dev/termination-log
|           |               /etc/hosts
`-nvme0n1p128 259:2   0     1M  0 part
nvme1n1    259:3   0    4G  0 disk /data
```

You can also see the PV that was created by the PVC:

```
$ kubectl get pv
pvc-f0a2ab76-9ac5-4334-9c8e-549066d88d8d    4Gi          RWO
Delete           Bound       default/ebs-claim    ebs-sc
<unset>                      10m
```

What happens to the PV when the Pod that claims it terminates? By default, Kubernetes will delete a dynamically provisioned PV when it is no longer in use. Using PV reclaim policy, you can also instruct Kubernetes to retain the volume, thereby decoupling the lifecycle of a Pod and a PV.

2.2.8 ConfigMaps

Consider that you have a bunch of apps running in your Kubernetes cluster, and each one needs some special settings to work properly - like a database connection string, or the logging level it should use. Instead of hard-coding all those details into your app's code, you can store them in a ConfigMap.

A *ConfigMap* is a Kubernetes resource that lets you maintain application configuration in one place, organized into handy key-value pairs. When you create a ConfigMap, you can define things like the port, log level, and other configuration for an application. When Kubernetes runs this application, it provides the configuration to the container as environment variables or a file. ConfigMap helps you externalize application configuration and decouple configuration from business logic.

ConfigMaps make it easy to update your app's settings without having to rebuild or redeploy the whole thing. If you need to change the database connection string, for example, you can just update the ConfigMap and your app will automatically start using the new value right away or when the container restarts.

Here's a minimal ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  my-key: my-value
```

Applications can access ConfigMaps key-value pairs as environment variable or files.

Listing 2.8: A Pod with a ConfigMap mounted as a volume

```
apiVersion: v1
```

```

kind: Pod
metadata:
  name: my-configmap-pod
spec:
  containers:
    - name: nginx-container
      image: nginx
      volumeMounts:
        - name: config-volume
          mountPath: /data #A
  volumes:
    - name: config-volume
      configMap:
        name: my-configmap #B
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap #B
data:
  my-key: my-value #C
#A Path where the ConfigMap is mounted
#B ConfigMap name
#C Application settings as key-value pair

```

By mounting the ConfigMap, we can read its contents from within the container. Every key in the ConfigMap is available as a separate file within the specified mount directory. You can view the ConfigMap inside a Pod by running:

```
$ kubectl exec my-configmap-pod -ti -- cat /data/my-key
my-value
```

ConfigMaps can also hold an entire file. For example, consider you have a container that runs a Python script. Instead of including the script in your container image, you can store the contents. This way, you can change the script by updating the ConfigMap and skip the process of rebuilding the container image.

Listing 2.9 Using kubectl to create a ConfigMap from a file's content

```

$ kubectl create configmap hello-world-py-script --from-
file=hello-world.py
configmap/my-py-script created

$ kubectl describe configmaps hello-world-py-script
Name:           hello-world-py-script
Namespace:      default
Labels:         <none>
Annotations:   <none>

Data

```

```
=====
hello-world.py:
-----
import time

while True:
    print("Hello world")
    time.sleep(5)
...

```

2.2.9 Secrets

Kubernetes Secrets provide you a place to store privileged data such as API keys, passwords, and tokens. Secrets are very similar to ConfigMaps, but since they are sensitive information, Kubernetes treats them differently. You can also choose to encrypt Secrets, which we highly recommend. The choice of encryption method depends on your cluster infrastructure. For example, Amazon EKS allows you to encrypt Secrets using a user provided AWS KMS Key.

Many cluster administrators choose to use a secrets storage service such as AWS Secrets Manager, Hashicorp Vault, GCP Secrets Manager. These services give you a central place for managing, accessing, and auditing secrets across your company. Open-source projects such as External Secrets Operator (<https://external-secrets.io/>) make it easier to inject secrets from external sources into apps running in a Kubernetes cluster.

2.3 Jobs

So far, we have dealt with types of containers that run continuously. Kubernetes also gives you to run containers that perform their function and stop. Kubernetes Jobs are a great fit for tasks, such as running data processing or scientific computations. When you create a Job, Kubernetes creates a Pod (or more) to run your container. The key difference between a Job and a regular Kubernetes Deployment is that a Job will run the task to completion and then stop, rather than continuously running the application.

With Kubernetes, you can parallelize your code to massive degrees. This parallelization becomes a necessity when the amount of data to be processed exceeds the processing capacity of a single server.

Listing 2.10: A job to output “Hello, world!”

```
apiVersion: batch/v1
kind: Job
metadata:
  name: print-hello-world
spec:
  template:
```

```

spec:
  containers:
    - name: print-hello-world
      image: busybox
      command: ["echo", "Hello, world!"]
    restartPolicy: OnFailure #A

```

#A The Pod will only be recreated if the container exits with a non-zero code.

Jobs find extensive usage in a machine learning system. They are the primary vehicle for running data processing, model training, and fine-tuning tasks. In the coming chapters, you'll become intimately familiar with them.

CRONJOBS

You can schedule jobs in your cluster using CronJobs. CronJobs in Kubernetes are like the traditional cron jobs found in Unix-based systems, but with a few key differences.

You schedule jobs by creating a CronJob resource. When the scheduled time arrives, Kubernetes creates a regular Job resource based on the CronJob's specification. This Job will then launch one or more Pods to execute the task.

Listing 2.11: A CronJob

```

apiVersion: batch/v1
kind: CronJob
metadata:
  name: backup-database
spec:
  schedule: "0 2 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: backup
              image: busybox
              command:
                - /bin/sh
                - -c
                - |
                  echo "Starting database backup..."
                  pg_dump my_database > /backup/database.sql
                  echo "Backup complete!"
    restartPolicy: OnFailure

```

2.3.1 StatefulSets

Earlier in the chapter, you learned how to add persistent storage to Pods using PVs and PVCs. Recall that in that scenario, if a Pod terminates, although the data stored in a PV is not lost, Kubernetes can attach any available PVs to a replacement Pod based on its claim. In other words, there are no guarantees that a replacement Pod gets the same PV as the Pod it is replacing.

However, when running stateful workloads, such as a MySQL database, Cassandra cluster, or Redis cache, you often need a stronger relationship between the compute (the Pod) and the storage (the PV). In Kubernetes, StatefulSets allow you to run workloads that require data persistence and ordinality.

StatefulSets are designed specifically for running stateful applications in Kubernetes. Unlike regular Deployments, which treat Pods as interchangeable units, StatefulSets provide each Pod with a unique and stable identity. This is crucial for stateful applications that require:

- Stable Network Identities – Each Pod in a StatefulSet is assigned a unique network identity, such as a DNS name, which remains the same even if the Pod is rescheduled or restarted.
- Persistent Storage – StatefulSets ensure that each Pod is associated with a specific Persistent Volume, preserving the data even if the Pod is terminated and replaced.
- Ordered Deployment and Scaling – StatefulSets guarantee that Pods are deployed and scaled in a specific order, ensuring that the application can handle the changes without disruption.

Listing 2.12 A sample StatefulSet with three replicas

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web #A
spec:
  serviceName: "nginx" #B
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.24.0-alpine
        volumeMounts:
          - name: www
            mountPath: /usr/share/nginx/html
  volumeClaimTemplates: #C
  - metadata:
      name: www
  spec:
    accessModes: [ "ReadWriteOnce" ]
```

```

resources:
  requests:
    storage: 1Gi
#A Name of the StatefulSet, which also determines Pod names
#B A headless service called nginx
#C Defines the properties of PV

```

When you created a Deployment, its Pods got a randomly generated name like my-deployment-75b877d8d6-cgjrq. Pods created by StatefulSets get a unique DNS name based on their ordinal index.

```

$ kubectl get pods
web-0      1/1     Running   0          28s #A
web-1      1/1     Running   0          20s #B
web-2      1/1     Running   0          11s
#A The first Pod in StatefulSet has index 0. Hence it's called
web-0
#B The second Pod is called web-1

```

2.3.2 DaemonSets

In the previous sections, you learned how Deployments and StatefulSets help you run scalable, replicated applications across a group of nodes. However, there are certain types of workloads that require a different scheduling approach. Imagine you have a system service, like a metrics collector or a log aggregator, that needs to run on every single node in your cluster. You don't want multiple replicas of this service - you want exactly one instance running on each node, so that you can collect data from all the nodes in a consistent way.

This is where Kubernetes DaemonSets come into play.

A *DaemonSet* is a Kubernetes resource that ensures one copy of a Pod is running on every (or a selection of) nodes in a Kubernetes cluster. Unlike Deployments or StatefulSets, which focus on scaling the number of replicas, DaemonSets are designed to ensure that a specific workload is running on every node.

When you create a DaemonSet, Kubernetes will automatically schedule a Pod that matches the DaemonSet's specification on each node in the cluster. As new nodes are added to the cluster, Kubernetes will automatically start a new Pod on those nodes. And if a node is removed, the corresponding Pod will be terminated.

This makes DaemonSets ideal for running system services, agents, or daemons that need to be present on every node, such as:

- Logging and monitoring agents (e.g., Fluentd, Prometheus Node Exporter)
- Network plugins (e.g., Calico, Cilium)
- Storage drivers (e.g., EBS, S3)
- Hardware management agents (e.g., NVIDIA GPU device plugin)

By using a DaemonSet, you can ensure that these critical system services are always running on every node in your Kubernetes cluster, without having to worry about manually scheduling or managing them.

2.4 Kubernetes Package Management

The preferred way of deploying workloads in a Kubernetes cluster is by creating YAML manifests for resources the application needs. This approach has several benefits:

- Human-Readable Configuration – The YAML manifests are written in a human-readable format, making it easy for developers, operators, and other team members to understand the application's deployment configuration. This transparency is crucial for maintaining and troubleshooting the application.
- Version Control and Auditing – By storing the YAML manifests in a code repository, you can leverage version control systems like Git to track changes, review history, and audit the application's deployment configuration over time. This is essential for maintaining a reliable and reproducible deployment process.
- Declarative Approach – Kubernetes follows a declarative model, where you define the desired state of your application in the manifests, and the Kubernetes control plane ensures that the actual state matches the desired state. This declarative approach makes it easier to manage the application's lifecycle, as you can simply re-apply the manifests to restore the application's state.
- Portability and Consistency – The YAML manifests can be shared, reused, and applied across different Kubernetes environments (e.g., development, staging, production), ensuring consistency and portability of the application deployment.
- Collaboration and Automation – By storing the YAML manifests in a code repository, multiple team members can collaborate on the application's deployment configuration, and the deployment process can be automated using tools like CI/CD pipelines.

One of the key factors that can limit the reusability of Kubernetes manifests is that the configuration is often very specific to a particular cluster or environment, with values hard coded in the manifests.

The challenge with hard-coded values in Kubernetes manifests is that it reduces the flexibility and portability of the application deployment. When the configuration is tightly coupled to a specific environment, it becomes difficult to

reuse the same manifests across different clusters, cloud providers, or even between development, staging, and production environments.

Kubernetes package managers such as Helm and Kustomize that improve the portability of Kubernetes workloads. They abstract the deployment configuration into reusable packages or overlays, simplifying the process of deploying applications across different Kubernetes clusters. They improve consistency by giving you the ability to define a "base" configuration and apply targeted customizations in different environments.

2.4.1 Helm

Helm is a package manager and templating engine for Kubernetes that allows you to package all the Kubernetes resources (Deployments, Services, ConfigMaps, etc.) required for an application into a single package called a "Chart". A chart is a bundle of pre-configured Kubernetes resources that define an application's deployment configuration.

Helm charts can be versioned, shared, and deployed across different Kubernetes clusters, regardless of their specific configurations. Helm's templating engine allows you to customize the deployment of an application based on different environments or user preferences, without the need to maintain multiple YAML files.

A Helm chart comprises templates and values files. Templates parametrize the definition of the Kubernetes resources while values files offer customization options for the template values. This approach allows you to create templates with replaceable placeholders for parameters that vary between different deployments or environments.

With Helm, you can deploy applications in one command. For example, here one simple command to deploy Airflow:

```
$ helm install my-airflow apache-airflow/airflow --version  
1.13.1
```

Summary

- Kubernetes is a cloud agnostic platform for running reliable and scalable distributed systems. It helps you deploy and manage workloads across a cluster of host machines (nodes).
- A cluster consists of a control plane and data plane. The control plane runs Kubernetes processes, and the data plane runs user workloads.
- Kubernetes provides support for multitenant environments. A cluster is logically partitioned into namespaces, which provide a way to organize and isolate resources within the same cluster.

- Kubernetes runs application in Pods, which run one or more containers on the same node. In practice, it's rare to create individual Pods as they are not fault tolerant. In production, applications are deployed by creating Deployments. Deployments enable you to deploy, scale, and upgrade a group of Pods.
- DaemonSets, StatefulSets, and Jobs give you more ways of running workloads. DaemonSets run once per node. StatefulSets establish tight coupling between compute and storage. Jobs are used to run workloads with finite runtime.
- Kubernetes Services give your applications a stable DNS name and IP address. They provide a way to load balance traffic across multiple Pods. Ingresses can expose multiple Services using a single load balancer.
- Kubernetes integrates with storage systems to support stateful workloads.
- Secrets and ConfigMaps give you a place to store sensitive information and application configuration, respectively.
- Kubernetes package management tools like Helm and Kustomize simplify distribution and deployment of Kubernetes application.
- GitOps allows you control your Kubernetes cluster and workload configuration using a Git repository. In this case, the Git repository becomes the single source of truth. All changes are tracked and triggered through pull requests.

3

Building an ML platform in Kubernetes

This chapter covers

- Setting up an Amazon EKS Kubernetes cluster using Terraform
- Creating an Ingress using NGINX Ingress Controller
- Deploy an identity provider using Keycloak
- Creating a scalable data science development environment using JupyterHub
- Enabling GPU workloads in Kubernetes

After learning the fundamentals of Kubernetes in the previous chapter, you are now ready to get your hands dirty. We'll need a real Kubernetes cluster for that. Deploying a production-grade Kubernetes cluster is quite a cumbersome task. Maintaining a Kubernetes cluster control plane is even more arduous. This is the reason most organizations use a managed Kubernetes service to reduce their operational burden. We will also use a managed Kubernetes cluster in this book. This book builds a data analytics and machine learning system on Amazon cloud.

While we're using AWS for the implementation, it is our goal to keep the architecture we propose in this book cloud-agnostic. Therefore, we won't be sacrificing our commitment to open-source tooling in the pursuit of cloud-native excellence.

To keep our promise of being cloud-agnostic, we have forced ourselves to limit the usage of managed services and chosen to deploy resources within the Kubernetes cluster wherever possible. While we'll adhere to our commitment to using open-source tooling for building the system, we'll use the following cloud specific services to build the environment and provide fundamental networking functions:

- Amazon EKS to create a Kubernetes cluster
- Amazon Route 53 for DNS

- AWS Certificate Manager to obtain a wildcard TLS certificate
- Elastic Load Balancer to provision load balancers
- Amazon Elastic Block Storage (EBS) for providing block storage
- Amazon Elastic File System (EFS) for providing NFS-based shared storage

While we have chosen these specific cloud services for this project, you are free to use alternative services that may be available in your own cloud or on-premises environment.

As we have established, an open-source ML system is composed of multiple interconnected components. Each component providing a specific functionality. Since there is no one tool or system that serves all the needs of an ML system, such a system is an amalgamation of tools that function together to provide the necessary capabilities data and ML scientists need to perform for data engineering and model development.

This chapter focuses on setting up the fundamental infrastructure required to run the system. Once you create the base infrastructure, we'll dive into building a scalable notebook environment that allows data scientists and ML engineers to perform data analytics and model development. We'll do this by deploying a JupyterHub environment on Kubernetes. The JupyterHub environment you'll create will have the following properties:

- Scalable Compute Resources – Leverage Kubernetes to dynamically allocate compute resources based on user demand, ensuring that each user has access to the necessary resources for their ML workloads.
- Secure Authentication and Authorization – Implement a robust authentication and authorization system using Keycloak to ensure that only authorized users can access the platform and that their actions are properly controlled based on their roles and permissions.
- Customizable Development Environments – Provide users with the ability to customize their development environments using Jupyter Notebook, allowing them to install the libraries, frameworks, and tools they need for their specific ML projects.
- Collaborative Workspaces – Enable users to share and collaborate on ML projects by providing shared workspaces and the ability to manage access control at a granular level.

In the upcoming chapters, we will show how to integrate this environment with other components of the ML toolchain, such as data storage, workflow orchestration, and experiment tracking.

The goal of this book is to equip you with the knowledge and skills needed to create a system that closely mirrors a scaled-down version of real-world

implementations. To that end, we will demonstrate how to build a system in the cloud without sparing any gory infrastructure details, as this is the deployment environment for most large-scale ML platforms.

3.1 Creating a Kubernetes cluster

Let's begin building. Before we can get going with showing you real world examples, we must get our hands on a Kubernetes cluster. If you don't currently have a Kubernetes cluster, you can create one in the AWS using the instructions in this book's Appendix. If you already have a Kubernetes cluster, you can reuse that cluster to deploy workloads.

Besides a Kubernetes cluster, you'll also need:

- The ability to create block storage volumes
- An NFS-based shared storage
- Virtual or physical machines to run Kubernetes nodes
 - Some machine workloads may require nodes with Nvidia GPUs
- Load balancing capabilities, to route traffic to applications running inside a Kubernetes cluster

Deployment steps included in this book assume that you're deploying an environment identical to the one described in the Appendix. If your environment differs, you may have to change some steps to suit your environment.

3.1.1 Creating an EKS cluster using Terraform

Infrastructure-as-Code (IaC) is a best practice for managing cloud resources using tools like Terraform. It offers significant benefits in repeatability and consistency by defining infrastructure through code. This approach ensures standardized deployments, reduces errors, and enables rapid scaling. Through automated provisioning and management, IaC improves efficiency and reduces effort in handling complex cloud infrastructures. Version control of infrastructure configurations allows better change tracking and easier rollbacks, improving security and recovery times.

Terraform, the IaC tool used in this book, allows you to define, provision, and manage cloud infrastructure using a declarative configuration language. By treating infrastructure as code, Terraform aligns with DevOps practices, enabling teams to apply software development principles to infrastructure management. It enables users to codify their infrastructure across multiple cloud providers, including AWS, Azure, and Google Cloud Platform, as well as on-premises environments.

This book provides Terraform code to deploy the base infrastructure in Amazon Web Services (AWS), upon which the rest of the MLOps platform is built. Detailed instructions for creating these resources are included in the Appendix. While AWS serves as the implementation platform for this book, it's important to note that all concepts are applicable across any cloud or hybrid environment. The principles and strategies you'll learn are equally relevant whether you're working with Microsoft Azure, Google Cloud Platform, or a private datacenter. The core requirement is the ability to run a Kubernetes cluster and access to appropriate hardware resources.

Whether you're using AWS or any other environment, you should have a Kubernetes cluster at this point to follow along. Your cluster should have the following capabilities:

8. It can autoscale worker nodes. If your environment doesn't support autoscaling, please add at least two virtual machines with at least 4 CPUs and 16 GB memory.
9. It can provision cloud load balancers. If you're operating in non-cloud environment, you'll have to manually create a load balancer such as MetalLB (<https://metallb.io>).
10. It can create block storage and NFS-based PersistentVolumes. In this book, block storage volumes are provided by Amazon EBS and NFS-based volumes are Amazon EFS filesystem. You can replace them with the storage services available in your environment.

So long as your cluster has these capabilities, you can implement the architecture presented in this book in your environment. We're not promising that you won't have to make any changes, but these will be limited to the underlying infrastructure. For instance, if you're operating in Google Cloud, you may create Persistent Disks instead of Amazon EBS volumes to get a block storage. But once you configure your cluster to provision block storage volumes using Google Cloud, you won't have to make changes to the code and manifests included in this book. This flexibility allows you to adapt the solutions to your preferred cloud provider while maintaining the core functionality described throughout.

3.2 Architecting an MLOps system on Kubernetes

From now on, this Kubernetes cluster will be our mechanism for infrastructure management. From this point onwards, we will use Kubernetes APIs to provision compute and storage. Thus, now we shift our focus towards setting up the core components of the MLOps system we want to build.

Let's document a few high-level requirements of this system:

1. It should cater to most data engineering and machine learning tasks such as running data pipelines, training machine learning models, and serving trained models.
2. It must hide infrastructure complexity from its end users.
3. It should be secure.
4. It should be scalable.
5. It should be observable. Its subcomponents must provide metrics, logs, and traces.

Figure 3.1 gives a high level overview of the open source tools that form the subcomponents of the Kubernetes-based MLOps system discussed in this book. We've intentionally over-simplified the diagram to keep the focus at a very high level. It will be premature to get into the intricacy of the plumbing of the system. Doing so will require Kubernetes and MLOps concepts we're yet to discuss. For now, let's just understand where these tools fit in the system:

1. Keycloak – Provides user authentication for web-based tools like JupyterHub.
2. JupyterHub – Multi-user platform for hosting and managing Jupyter Notebook environments.
3. Apache Airflow – Workflow orchestrator for managing and running data pipelines.
4. Apache Spark – Analytics engine designed for large-scale data processing.
5. Kubeflow – Kubernetes Operators with various MLOps capabilities, such as distributed training, model serving, ML pipelines, and fine tuning.
6. Ray – Distributed computing framework for Python applications.
7. Prometheus – Monitoring and alerting toolkit.
8. Grafana – Highly customizable tool for data visualization and monitoring.

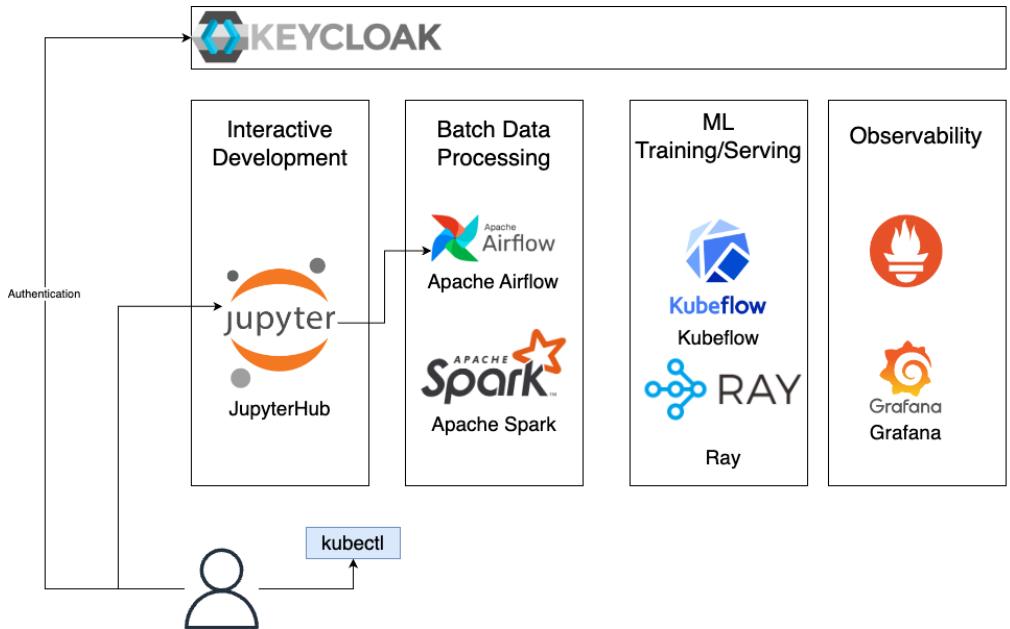


Figure 3.1 Overview of the MLOps platform. The platform provides end-to-end functionality for developing and operating applications with machine learning capabilities.

If you don't understand terms like "monitoring and alerting toolkit" or "Kubernetes Operator", fret not. These topics are discussed in detail later in the book. For now, we're concerned about the next milestone, which is building a multi-user Jupyter Notebook environment. This environment will allow users to analyze data and build models using interactive development environment that is managed for them.

Who are these users? For the purposes of this learning exercise, let's pretend we're building an MLOps system for a medium-sized bank. Currently the organization has a few machine learning and data scientists operating independently, exploring ML capabilities. We estimate that soon there will multiple data analytics and machine learning teams in need of a platform to run scalable data pipelines, model training, model hosting, and similar capabilities. To provide these capabilities, we decide to build a system on Kubernetes.

Who are we? We are the folks responsible for the infrastructure. Our mission is to ensure data and ML engineering teams have the tools and computing resources they require to build ML systems. We make these engineering teams successful by managing the Kubernetes-based platform. Additionally, we provide

abstractions that enable users to run jobs without having to understand any aspect of the underlying infrastructure.

By the end of the current milestone, data and ML engineering team members in our organization will be able to get a Jupyter Notebook in the cloud. This environment (that we manage) will give them (data and ML engineers) a place to run resource-intensive tasks like a Spark job or model training without being limited by the hardware on their local device.

Let's kick off this milestone by deploying Keycloak, which will allow us to provide identity to the users of the MLOps system. This identity subsystem will ensure that users (data and ML engineers) can access these tools securely.

3.3 Setting up an identity provider

To build a multi-user system, we need a system that provides user identity. We need a system to differentiate administrators from users. One that allows permitted users seamless access to the system.

Most organizations already use tools like Active Directory, Lightweight Directory Access Protocol (LDAP), or a similar technology for managing users and system access. Enterprise tools like email, shared drives, and printers integrate with identify systems to maintain uniformity across systems and centralize access controls. Unfortunately, our current environment doesn't have a user identity system yet.

Most open source MLOps tools provide web and API interfaces for end users and administrators. Many of these tools don't have an in-built identity and access management but rely on external services to provide this functionality. They support user authentication through a variety of open standards, including OAuth 2.0 and OpenID Connect protocols, allowing integration with popular identity providers.

We will implement a centralized user identity system using Keycloak. The MLOps tools (like JupyterHub) we deploy later will authenticate users through Keycloak. Before accessing an application, users will first log in to Keycloak. Upon successful authentication, Keycloak will redirect the user to the requested tool. This process is like accessing Gmail when not logged into Google, where you are redirected to Google.com for authentication before gaining access to your email.

Keycloak is an enterprise-grade, open-source identity and access management solution that provides features such as single sign-on (SSO), user federation, and role-based access control (RBAC). It supports various authentication protocols, including OpenID Connect, OAuth 2.0, and SAML 2.0, making it compatible with a wide range of applications and services.

By integrating an identity provider like Keycloak with your ML tools, you can:

- Centralize authentication – Users can access all the ML tools in your platform using a single set of credentials, eliminating the need to manage separate accounts for each tool.
- Enforce access control – Define roles and permissions in Keycloak to control user access to specific resources and actions within your ML tools, ensuring that users can only access what they are authorized to.
- Integrate with existing identity management – Keycloak can federate with your organization's existing identity management system, such as Active Directory or LDAP, allowing you to reuse existing user accounts and groups.
- Enable single sign-on (SSO) – Users can authenticate once and access multiple ML tools without being prompted for credentials each time, providing a seamless user experience.

If your organization already has an identity provider, then you may not need Keycloak when you architect a system in real life. If your identity provider is unusable for whatever reasons, you can explore using Keycloak to add support for additional authentication mechanisms. Keycloak can work in conjunction with your organization's existing identity management system or as a standalone identity provider for your ML platform. In this book, it acts as a standalone identity provider.

3.3.1 Configuring DNS

Our MLOps platform will offer a suite of tools accessible through web browsers. To enhance user experience, we'll assign easy-to-remember addresses for these tools. This approach streamlines access and improves productivity for our users. For instance, to access a Jupyter Notebook in the cloud, all they must do is open `jupyterhub.mycompany.com` in their browsers.

You'll need a registered domain for this. You can either get a new public domain for this or use one that your organization provides. If you don't already have a public domain, you can purchase one from Route 53 (<https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/registrar.html>) or another registrar like GoDaddy. You can also use free domain name service. Please ensure that the registrar you choose allows you to change name servers, so you can manage it using Route 53.

For this book, the authors purchased “`mlopsbook.online`” domain from GoDaddy for less than two dollars. To use this domain with Route 53, we changed the domain’s name servers to a Route 53 hosted zone name servers. The book will reference this domain name in sample code and illustrations.

Store your registered domain in an environment variable. We'll reference it in subsequent commands:

```
$ export DOMAIN=mlopsbook.online
```

Next, you'll need a Route 53 public hosted zone so you can create easy to remember addresses for various tools you'll use to build your ML platform. For example, when users need a Jupyter notebook, they can go to <https://platform.mlopsbook.online/jupyter> and start coding.

Create a public hosted zone in Route 53:

```
$ aws route53 create-hosted-zone \
  --name ${DOMAIN} \
  --caller-reference $(date +"%Y%m%d%H%M%S")
{
  "Location": "https://route53.amazonaws.com/2013-04-
01/hostedzone/Z08293792AN6EXBASDLKR",
  ...
  "DelegationSet": {
    "NameServers": [
      "ns-XXXX.awsdns-12.co.uk",
      "ns-XXXX.awsdns-44.net",
      "ns-XXXX.awsdns-60.org",
      "ns-XXXX.awsdns-40.com"
    ]
  }
}
```

If you didn't register your domain using Route 53, note the name servers shown in the output of `create-hosted-zone`. You'll configure them as the name servers with your registrar.

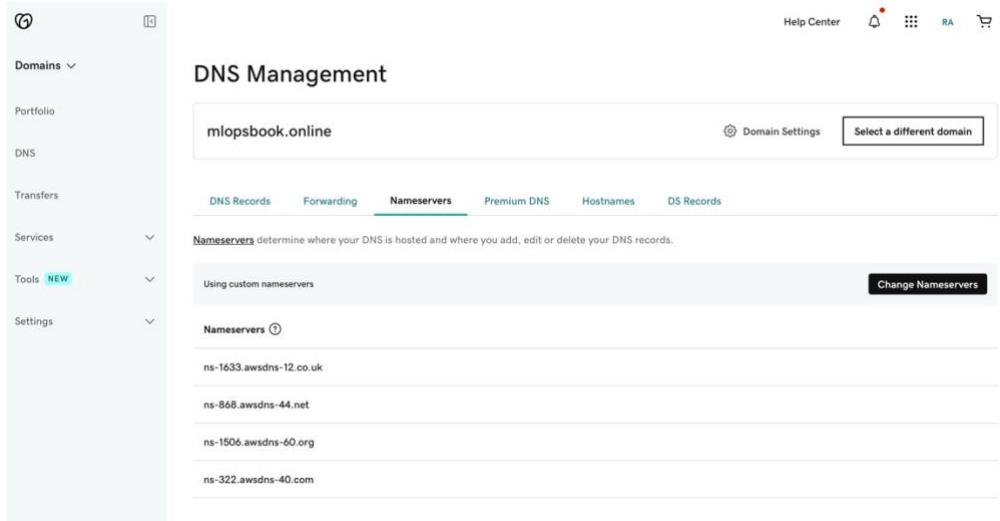


Figure 3.2 When using a domain that's not registered with Route 53, configure the domain's name servers to point to your Route 53 hosted zone's name servers.

To verify that an externally purchased domain uses Route 53 name servers, you can use the `dig` command (on Windows, use `nslookup -type=NS %Domain%`):

```
$ dig +short NS $DOMAIN
ns-1633.awsdns-12.co.uk.
ns-322.awsdns-40.com.
ns-868.awsdns-44.net.
ns-1506.awsdns-60.org.
```

If the name servers have “awsdns” in their names, your domain name uses Route 53 name servers.

3.3.2 Getting a TLS certificate

We'll expose ML tools and services in this cluster to users via web-based user interfaces. To provide secure connectivity between user's browsers and these tools, we'll use a Transport Layer Security (TLS) certificate. We'll use AWS Certificate Manager (ACM) to obtain this certificate since public TLS certificates provisioned through ACM are free.

To create the certificate, run the `get-acm-cert.sh` script included in the book's git repository. This script automates the process of requesting and validating an AWS ACM (AWS Certificate Manager) certificate using DNS validation. It first requests a certificate for a given domain (and its wildcard

subdomain), retrieves the required CNAME record details for DNS validation, and identifies the Route 53 hosted zone ID for the domain. It then creates or updates the necessary CNAME record in Route 53 to complete the DNS validation process. The script waits until the certificate is validated and issued, finally outputting the certificate's status and ARN.

```
$ sh ./get-acm-cert.sh $DOMAIN
CNAME_NAME is empty. Retrying in 5 seconds...
[ACM]           Waiting for certificate to be validated...
ISSUED
arn:aws:acm:us-west-2:01234567890:certificate/ac5be87c-b4a5-
4a65-b405-1f8eef1752d
```

The script outputs the Amazon Resource Number (ARN) of the certificate. An *ARN* is a unique identification of a cloud resource in AWS. Let's store it in an environment variable, as we'll use it later when deploying other applications.

```
$ export ACM_CERT_ARN=<The ARN of the certificate. For example,
arn:aws:acm:us-west-2:01234567890:certificate/**>
```

3.3.3 Deploying an Ingress Controller

In chapter 2, you learned about Kubernetes Ingresses. In short, they allow you to expose multiple web services using a single load balancer. There are many open source and commercial ingress controllers in the market. We'll use NGINX Ingress Controller (<https://github.com/kubernetes/ingress-nginx>) to expose web applications like JupyterHub and Keycloak.

The *NGINX Ingress Controller* is a load balancer for Kubernetes. In the cloud, you can attach a single Layer 4 load balancer (like a Network Load Balancer on AWS) and use it to route traffic to multiple services running inside a Kubernetes cluster. By using a single load balancer to route traffic to multiple services, the NGINX Ingress Controller significantly reduces infrastructure costs and simplifies management. This is particularly advantageous in cloud environments where each load balancer incurs separate charges.

At the core of Ingress functionality are Ingress rules, which define how incoming traffic should be directed to backend services. These rules can be based on various criteria such as hostnames, paths, or even custom headers. For example, an Ingress rule might specify that requests to "example.com/api" should be routed to a backend API service, while requests to "example.com/app" are directed to a frontend application service.

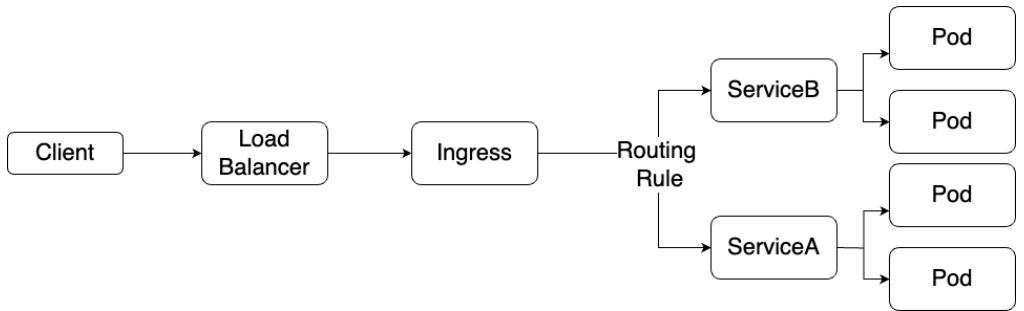


Figure 3.3 Ingress acts as a gateway, exposing HTTP and HTTPS routes from outside the Kubernetes cluster to internal services, with traffic routing governed by rules defined on Ingress resources within a cluster

Load balancing is a key feature of Ingress, enhancing application performance and availability. By distributing incoming traffic across multiple backend pods, Ingress ensures that no single pod becomes a bottleneck. This supports dynamic scaling, where additional pods can be added without user intervention. Ingress controllers, which are responsible for implementing the Ingress rules, often provide both layer 7 (application layer) and layer 4 (transport layer) load balancing capabilities.

Ingress also offers advanced features like SSL/TLS termination, offloading encryption and decryption tasks from application services to the Ingress layer. This not only improves security but also simplifies certificate management across multiple services. Additionally, Ingress can implement centralized access control, allowing administrators to define and manage traffic policies at a single point rather than configuring separate rules for each service.

Using Helm, deploy the NGINX Ingress Controller using the `nginx-values.yaml` file in the `nginx` directory:

```

$ cd ../nginx
$ envsubst < nginx-values.yaml.template > nginx-values.yaml
$ helm repo add ingress-nginx
https://kubernetes.github.io/ingress-nginx
$ helm repo update
$ helm upgrade -i ingress-ingress nginx-ingress/ingress-ingress \
--version 4.12.0 \
--namespace kube-system \
--values nginx-values.yaml

```

Once installed, the NGINX Ingress Controller creates a Load Balancer Service in the `kube-system` namespace. As soon as this Service gets created, the AWS Load Balancer Controller (<https://github.com/kubernetes-sigs/aws-load-balancer-controller>)

controller), which runs on the control plane in EKS auto mode, will provision a Network Load Balancer. This load balancer listens for traffic on ports 80 and 443 and has TLS configuration using the ACM certificate you created earlier.

What if your cluster isn't running on Amazon EKS? If you're operating in Google Cloud or Azure, your Kubernetes cluster will have a different Kubernetes controller that provisions and configures load balancer services available in that environment. The concept remains unchanged. So do the series of steps. When you install the NGINX Ingress Controller it creates a Kubernetes Service. In response, some cloud-specific controller provisions and configures the load balancer to forward traffic to NGINX Pods. Users access the application by going to the DNS name attached to the load balancer. The load balancer then forwards the traffic to NGINX Pods. NGINX inspects the web traffic and routes it based on the request's properties, such as header, path, or host.

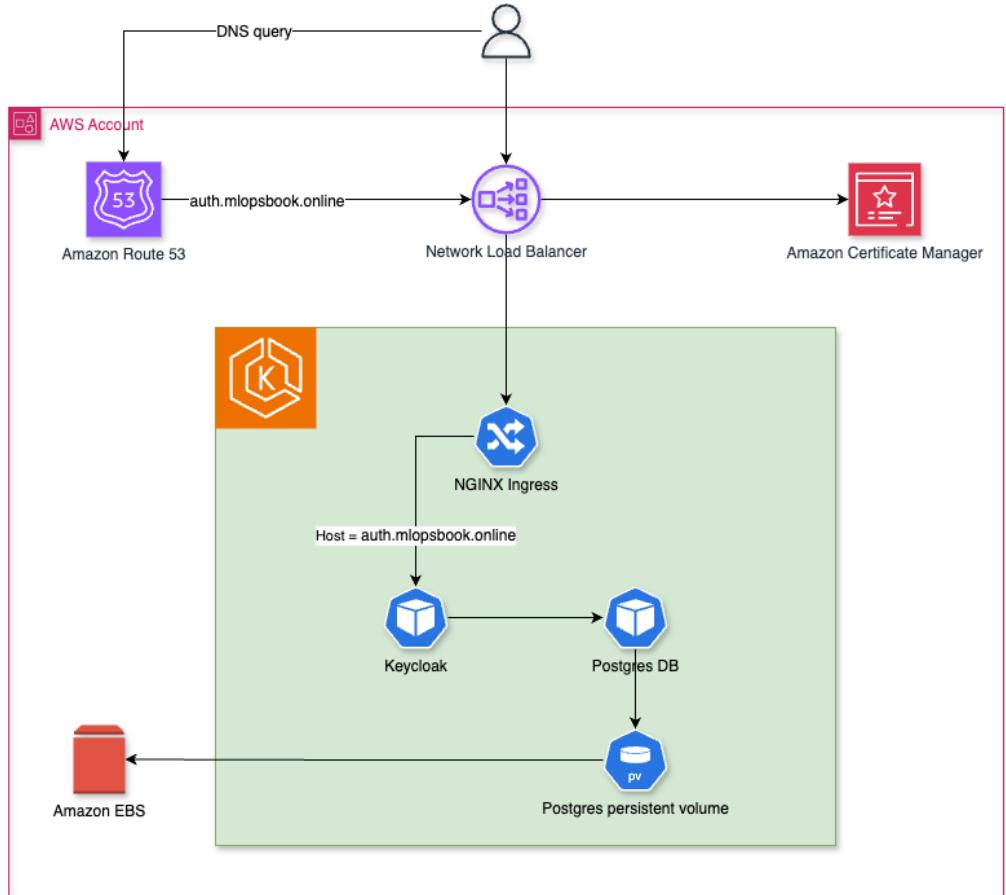


Figure 3.4 Schematic diagram of Keycloak deployment. The NGINX Ingress routes traffic for host=`auth.mlopsbook.online` to the Keycloak Pod. Keycloak stores its data in a Postgres database, which runs as a StatefulSet. The database state is persisted on a PV backed Amazon EBS.

As traffic hits the NGINX process running inside Kubernetes Pods, NGINX uses the certificate we created earlier to decrypt requests and route it to downstream applications. Currently, we only have a single NGINX replica accepting and routing external traffic. This configuration wouldn't be ideal in production where single points of failure should be avoided.

In production, you'd definitely want to have multiple NGINX replicas to minimize downtime and ensure high availability. By running multiple replicas,

you can distribute the load across several instances, improving performance and resilience. If one replica fails or needs to be updated, the others can continue to handle incoming traffic seamlessly. This setup also allows for rolling updates, where you can gradually replace old replicas with new ones without interrupting service.

NOTE It is a security best practice to rotate TLS certificates regularly. In production, please consider implementing a method to auto-renew certificates. Depending on your certificate issuer, you may be able to use open source tools like cert-manager (<https://cert-manager.io>) to the renewal process.

Moreover, in a production environment, you would likely implement additional security measures and optimizations. This could include setting up Web Application Firewall (WAF) rules within NGINX to protect against common web vulnerabilities, implementing rate limiting to prevent abuse, and fine-tuning caching strategies to improve response times. You might also consider using external load balancers or cloud provider-specific ingress controllers that offer advanced features like automatic scaling based on traffic patterns.

The first service we'll expose using NGINX will be the identity provider. This service will be exposed at `https://auth.<domain>.<tld>` (for example `https://auth.mycompany.com`). Whenever NGINX receives traffic for this DNS name, it will forward traffic to the identity provider's Pod.

3.3.4 Creating an Identity Provider using Keycloak

Keycloak will provide user authentication functionality in this platform. We've chosen Keycloak because its open source and has a wide adoption. You should know that there are other self-hosted and SaaS-based alternatives to Keycloak, and depending on your organization's capabilities, you may not need to create an identity provider at all. You can swap Keycloak with any other identity provider as long as it provides standard authentication mechanisms like OAuth.

Alternatively, you can use Keycloak as a bridge between your identity provider and client applications. In this scenario, user identity is managed in an external centralized system and Keycloak acts a proxy for authenticating users.

For now, we'll assume our fictional organization has no existing identity provider. All user identities are managed in Keycloak. To deploy Key, we'll create a namespace called "keycloak" in our cluster:

```
$ kubectl create namespace keycloak
```

Recall that namespaces are way to group resources within a cluster. We're creating a new namespace for Keycloak so any components that belong to Keycloak are "inside" this namespace. What could these components be? They are Kubernetes resources like Deployments, Services, ConfigMaps, and Persistent Volumes.

Keycloak requires a PostgreSQL database. For our initial setup, we'll allow the Keycloak Helm chart to create a new database with default settings. However, for production environments, we strongly recommend using a highly available PostgreSQL cluster to ensure Keycloak's continued functionality in case of database failures. For our current needs, a single database replica will suffice.

Generate a HELM values file and install Keycloak:

```
$ cd ../keycloak
$ envsubst < keycloak-values.yaml.template > keycloak-
values.yaml
$ helm upgrade --install kc \
oci://registry-1.docker.io/bitnamicharts/keycloak \
--values keycloak-values.yaml \
--version 21.2.2 \
--namespace=keycloak
```

You'll now have two StatefulSets running in the `keycloak` namespace. One for Keycloak itself and another for Postgres database, which is required for Keycloak.

```
$ kubectl -n keycloak get pods
NAME           READY   STATUS    RESTARTS   AGE
kc-keycloak-0  1/1     Running   0          2m1s
kc-postgresql-0 1/1     Running   0          2m1s
```

WARNING For demonstration and ease of access, we have exposed the Keycloak admin dashboard to the public internet. In production, the admin dashboard should only be available from the organization's private network. In this case, Keycloak's `/admin` and `/auth` URLs will be different.

Next, you'll create an A Name record so the load balancer is accessible at the `https://auth.<yourdomain.com>`.

If you're deploying in AWS, we've shared a script on GitHub to automate the process of adding an alias A record in Amazon Route 53 for a specified domain and subdomain. It first retrieves the hosted zone ID for the given domain using AWS CLI. Then, it fetches the NLB address of the NGINX ingress controller. Finally, it constructs a change batch JSON object and creates an alias A record

that points the specified subdomain to the NLB. This script effectively tells Route 53 to route traffic bound to http://auth.<yourdomain.com> to the load balancers attached to the ingress controller we deployed earlier.

```
$ sh create-a-record.sh $DOMAIN auth
```

NOTE Instead of creating DNS entries manually, you can automate DNS configuration using ExternalDNS (<https://github.com/kubernetes-sigs/external-dns>), which synchronizes exposed Kubernetes Services and Ingresses with DNS providers.

If you aren't in AWS, please create an A record with your DNS service provider. Point the A record to the IP or DNS name of your load balancer.

Once DNS changes have replicated, which usually takes two to five minutes, you can access the Keycloak administration console at https://auth.<yourdomain.com>. Login with username and password "mladmin". The admin dashboard is publicly accessible, we recommend changing the password immediately to avoid unauthorized access.

WARNING At the time of writing, there's a bug in Keycloak that prevents the "manage account" page from loading. If you get an error message stating "failed to initialize keycloak", please follow the instructions in this thread: <https://keycloak.discourse.group/t/the-account-console-presents-failed-to-initialize-keycloak-init-request-returns-403/8918/6>.

With this you have deployed Keycloak, which you will use to authenticate users and provide secure access to the tools you'll use to develop and deploy ML applications.

3.3.5 Preparing Keycloak for client authentication

Now that you've installed Keycloak in your Kubernetes cluster, you're ready to configure it to authenticate users. The first users we will authenticate are the users in our fictional organization that need access to JupyterHub. These will primarily be engineers working in data analytics and machine learning roles. Users in this job profile often use tools like JupyterHub to run interactive Notebooks for data analysis. We will simplify the process of how a user in the organization gets a pre-configured Jupyter Notebook environment.

We'll setup the system in such a way that when a user needs a Jupyter notebook environment to perform data analysis, all they need is a browser and network connection. But we want to ensure that only authenticated users have

access to such an environment. To implement this user authentication, we use Keycloak.

Later in this chapter when we deploy JupyterHub, we'll configure it so it authenticates users before logging them in. JupyterHub will redirect unauthenticated users to Keycloak. Once authenticated, Keycloak will return the user to JupyterHub.

We begin with creating a client for JupyterHub. A *client* in Keycloak is an entity that interacts with Keycloak to authenticate users and obtain tokens. Clients are applications or services acting on behalf of users to provide a single sign-on experience.

Log in to the Keycloak admin console and select Clients in the left navigation panel. Next, click the Create Client button.

Clients > Create client

Create client

Clients are applications and services that can request authentication of a user.

1 General Settings 2 Capability config 3 Login settings

Client type	OpenID Connect
Client ID *	jupyter
Name	jupyter
Description	Client for Jupyter Notebook user authentication

Always display in UI Off

Next Back Cancel

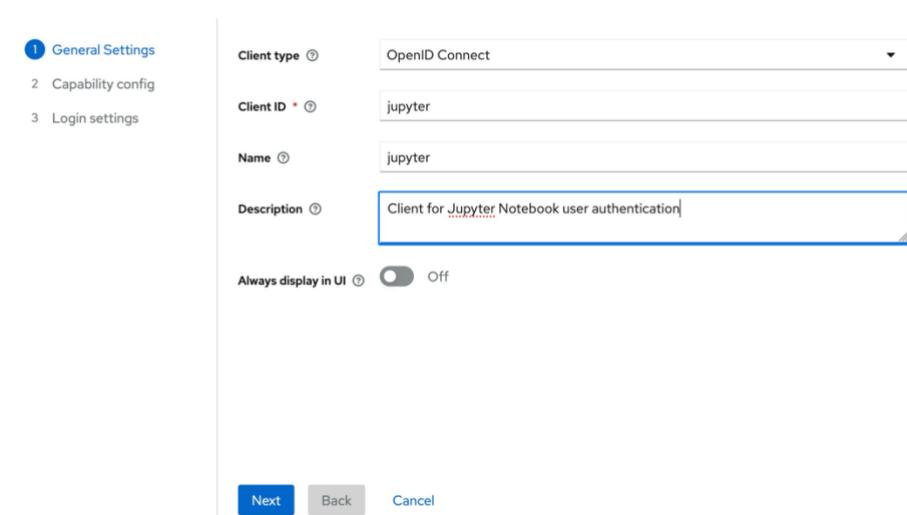


Figure 3.5 Creating a Jupyter client in Keycloak to authenticate Jupyter notebook users

In the next screen, enable “Client authentication” and “authorization”.

Create client

Clients are applications and services that can request authentication of a user.

1 General Settings

2 Capability config

3 Login settings

Client authentication On

Authorization On

Authentication flow

Standard flow

Direct access grants

Implicit flow

Service accounts roles

OAuth 2.0 Device Authorization Grant

OIDC CIBA Grant

Next Back Cancel

Figure 3.6 Enabling client authentication enables users accessing Jupyter to sign on using Keycloak credentials.

The next screen prompts you to enter URLs for the client you're creating. Since this client is for JupyterHub, it will be helpful for developers to have an easy to remember address. For example, we may set up the system so Jupyter users can access notebooks at <https://platform.mlbook.online/jupyter>. Enter the URLs to match your domain in the "Login Settings" page:

Create client

Clients are applications and services that can request authentication of a user.

1 General Settings	Root URL ⓘ	https://platform.mllopsbook.online
2 Capability config	Home URL ⓘ	https://platform.mllopsbook.online/jupyter
3 Login settings	Valid redirect URLs ⓘ	https://platform.mllopsbook.online/* Add valid redirect URLs
	Valid post logout redirect URLs ⓘ	https://platform.mllopsbook.online/* Add valid post logout redirect URLs
	Web origins ⓘ	Add web origins

[Save](#) [Back](#) [Cancel](#)

Figure 3.7 Screenshot of Jupyter client login settings in Keycloak admin dashboard showing URL configurations for Jupyter.

Next, navigate to “Credentials” tab in client settings and click on the copy button right next to “Client secret” to copy the secret, you’ll need it later while setting up JupyterHub to authenticate with Keycloak. The client secret is the password a client uses to prove its identity. It should be treated as sensitive data.

The screenshot shows the 'Clients > Client details' section in Keycloak. The client is named 'jupyter' and is associated with 'OpenID Connect'. The 'Enabled' switch is turned on. The 'Credentials' tab is selected, showing two sections: 'Client Authenticator' (set to 'Client Id and Secret') and 'Client secret' (containing a masked value). A 'Copy to clipboard' button is overlaid on the 'Client secret' input field. Below these are sections for 'Registration access token' and 'Refresh token'.

Figure 3.8 Screenshot showing user credential tab in Keycloak. A client secret in Keycloak is a credential used by confidential clients to authenticate with the Keycloak authorization server when requesting access tokens.

Store the client secret in an environment variable:

```
$ export JUPYTER_CLIENT_SECRET=sdI38skjsfp6r7Znmui5Sn7Lr7t9sWs
```

This step finishes the configuration for Keycloak to support JupyterHub authentication. Note that in the current setup, there's only one Keycloak Pod handling all operations. In production environment, you should deploy multiple Keycloak replicas for high availability and load balancing. We recommend reviewing Keycloak best practices for production deployments available here: <https://www.keycloak.org/server/configuration-production>.

Keycloak provides excellent support for federating with existing identity providers (IdPs) in your organization. This allows you to leverage your existing user stores and authentication mechanisms while still benefiting from Keycloak's advanced features and capabilities.

3.3.6 Create a user in Keycloak

So far you have been using Keycloak as an administrator. Now, we'll create an unprivileged user in Keycloak that will have access to login to the ML tools you'll deploy later.

Navigate to Users the left navigation panel in Keycloak and:

- 11.create a user called "mluser" with first and last name set to mluser. Be sure to set the names because applications like Apache Airflow, which we'll use later in the book, require users to have first and last names defined. You can leave other settings to their default values.
- 12.Next, navigate to the Credentials tab and create a password for this user.

From this point on, we'll use the "mluser" user for all tasks other than administering Keycloak itself. We'll reserve the usage of the "mladmin" user for Keycloak administration tasks.

3.3.7 Understanding the authentication workflow

Keycloak provides authentication for applications allowing users to log in to different applications using a single set of credentials such as a username and password. Keycloak also enables SSO. Once a user logs in to one application, they can access other applications without having to sign in again.

Once you've created a client in Keycloak, users can login to the client application using their Keycloak credentials. When a user logs in to a client application, the client application redirects the user to Keycloak login page. The client application also includes the client ID and secret when redirecting to Keycloak.

Upon successful authentication, Keycloak generates an authorization code and redirects the user back to the client application with the authorization code. The client application receives the authorization code and sends a request to Keycloak's /token endpoint to exchange the authorization code for an access token and an ID token.

Keycloak's /token endpoint validates the authorization code, issues tokens, and send them back to the client application. The client application then uses the access token to request user information from Keycloak's /userinfo endpoint. Keycloak responds with user's profile information. At this point, the client application establishes a session for the user based on the retrieved information.

You can obtain the URLs of Keycloak endpoint by accessing the Keycloak "OpenID Endpoint Configuration" API endpoint. The address of the endpoint follows this schema:

<https://<Keycloak DNS name>/realms/<realm name>/.well-known/openid-configuration>. The Keycloak admin dashboard has a link to this page under “Realm Settings”.

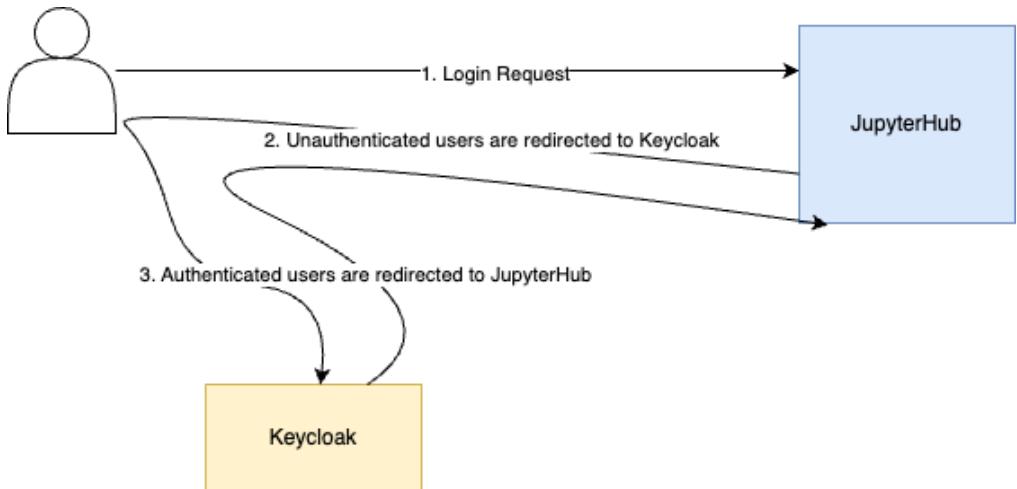


Figure 3.9 The sequence of interactions between the user, client application, and Keycloak during the authentication process. When an unauthenticated user accesses JupyterHub, they are redirected to Keycloak for authentication. Upon successful authentication, users are redirected to JupyterHub.

3.4 Creating a self-service development environment

To maximize developer productivity and eliminate manual processes that hinder access to the tools they need, we'll create a self-service system for ML and data scientists to provision development environments on-demand. In this section, we'll explore deploying JupyterHub (<https://jupyter.org/hub>) in your Kubernetes cluster and using Keycloak for user authentication.

JupyterHub on Kubernetes offers a scalable solution for multi-user environments, particularly for ML workloads. By leveraging Kubernetes orchestration capabilities, JupyterHub can dynamically provision isolated Jupyter Notebook instances as Pods for each user. This ensures that every data scientist or ML engineer has access to a dedicated environment with guaranteed CPU, memory, storage, and GPU resources. Kubernetes resource quotas and autoscaling features allow organizations to optimize resource utilization, scaling compute power up or down based on demand. Additionally, persistent storage ensures that users' work is saved even if their notebook pods are terminated or rescheduled.

By providing a self-service platform, ML and data scientists can quickly provision notebook environments they need, without waiting for manual provisioning or approvals. As a bonus to this deployment method, you'll also standardize the initial state of the development environment. Since every notebook originates from the same container image, you ensure consistency across all development environments.

3.4.1 Deploy JupyterHub

We'll deploy JupyterHub in its own Kubernetes namespace. Namespaces not only help you logically group resources, but they can also be used to isolate workloads in Kubernetes. Furthermore, you can implement quotas (<https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/quota-memory-cpu-namespace/>) to ensure that workloads running in a namespace don't consume all available CPU and memory resources in a cluster, starving workloads running in other namespaces.

First, setup environment variables needed for setting up JupyterHub:

```
$ export JH_HOSTNAME=platform.${DOMAIN}
$ export KEYCLOAK_HOSTNAME=auth.${DOMAIN}
$ export JH_VERSION=3.3.5
```

Go to the Chapter 3/jupyterhub directory and generate a Helm values file using the template:

```
$ cd ../jupyterhub
$ envsubst < jupyterhub-values.yaml.template > jupyterhub-
values.yaml
```

Install JupyterHub using HELM:

```
$ helm repo add jupyterhub https://hub.jupyter.org/helm-chart/
$ helm repo update
$ helm upgrade --cleanup-on-fail \
--install jhub jupyterhub/jupyterhub \
--namespace jupyter \
--version=${JH_VERSION} \
--values jupyterhub-values.yaml
```

Next, we must configure DNS so we can access JupyterHub using an easy to remember URL. To do this, we created a record in the Route 53 hosted zone, so when a user needs a Jupyter notebook, they can simply open their browsers and go to https://platform.<your_domain>.<tld>/jupyter.

```
$ sh create-a-record.sh $DOMAIN platform
```

When you go to that URL (for example, <https://platform.mlopsbook.online/jupyter>), you'll be presented with a login page.

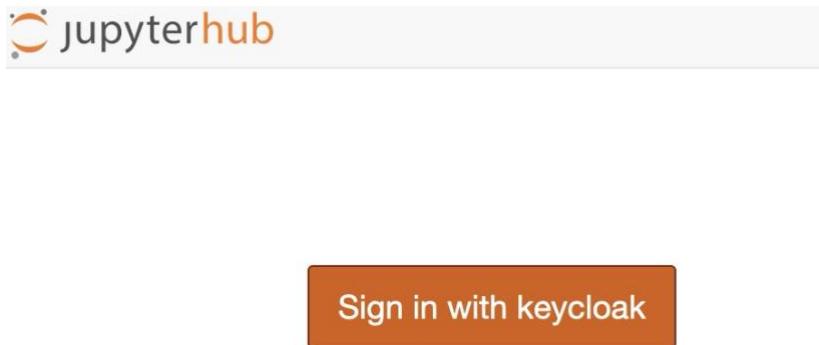


Figure 3.10 Screenshot of JupyterHub login page configured to use Keycloak to offload user authentication.

You can now login as username "mluser" and the password you had set for the user in Keycloak. If you are already logged Keycloak as the "mladmin" user, you'll need to logout of the current Keycloak session.

3.4.2 Role-Based Access Control with Keycloak

Keycloak's user management capabilities are particularly valuable when building an environment that needs to support a diverse range of developers with varying roles and responsibilities. One of the key features that Keycloak offers is role-based access control (RBAC), which allows you to grant users access to applications based on their job functions while preventing unauthorized access.

With RBAC, you can define roles that represent different levels of access or responsibilities within your organization. For example, you might have roles such as "Developer," "Project Manager," "Data Scientist," or "Administrator." Each role can be assigned a set of permissions that determine the applications, resources, or actions that users with that role are allowed to access or perform.

Once you have defined the roles and their associated permissions, you can assign these roles to individual users or groups of users within Keycloak. This process is typically straightforward and can be done through Keycloak's user interface or programmatically via its APIs. For instance, you might assign the "Developer" role to a team of software engineers, granting them access to the integrated development environment (IDE), version control system, and deployment tools. At the same time, you could assign the "Data Scientist" role to a group of data analysts, allowing them to access data processing and analysis tools while restricting their access to the codebase.

3.4.3 Providing persistence to notebook servers

Let's bring our attention back to JupyterHub. When a user signs into JupyterHub for the first time, JupyterHub creates a new notebook Pod based on a predefined template. This Pod serves as the user's personal notebook server. If the user disconnects from the session and logs in again later, JupyterHub will reconnect them to their existing notebook server.

To ensure users don't lose their work and data in the notebook workspace, each user's Pod is assigned a persistent volume backed by a storage service like Amazon EBS (<https://aws.amazon.com/ebs/>). In other cloud and hybrid environments you can use a similar storage service. If a user's notebook server crashes or moves to another node, Kubernetes will reattach the persistent volume, preventing any data loss during infrastructure failure events.

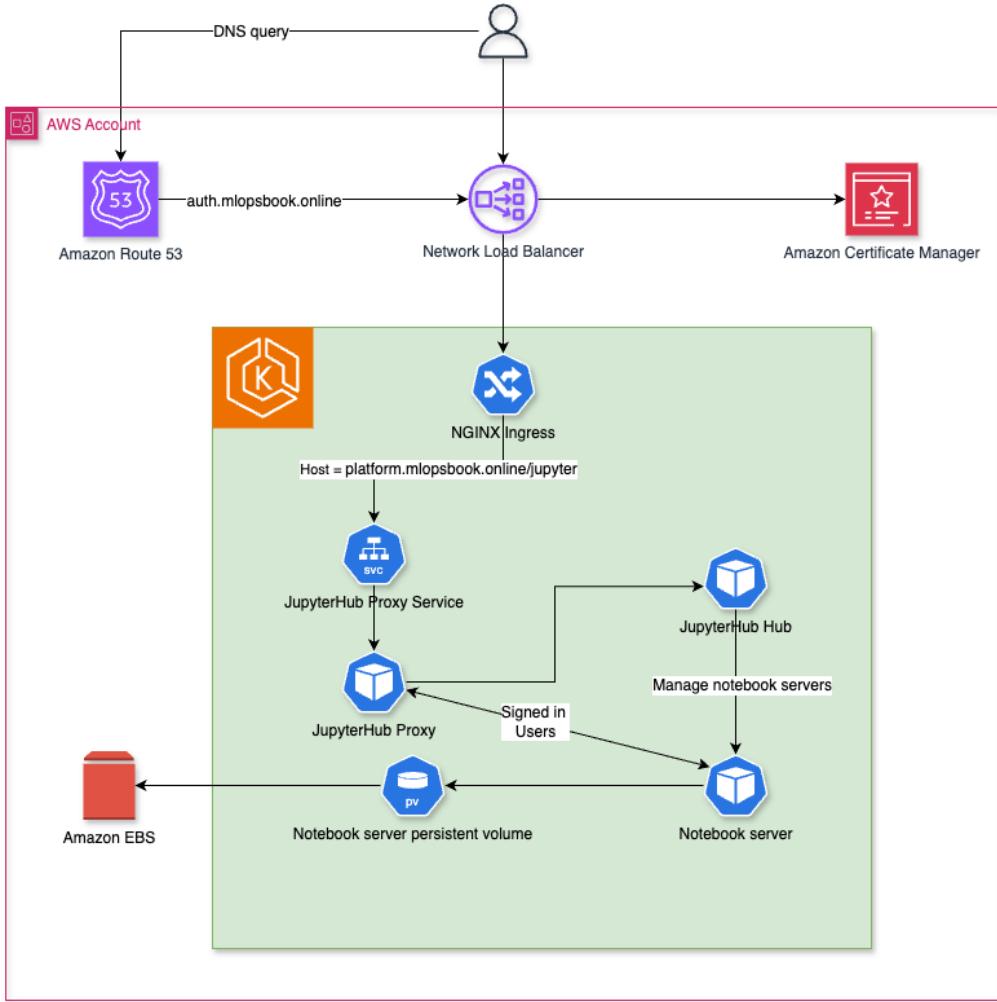


Figure 3.11 The NGINX Ingress forwards the traffic to the JupyterHub Proxy Service, which acts as an intermediary between the users and notebook server. Each notebook server is connected to a persistent volume, which is an Amazon EBS volume. This ensures that user data is stored persistently and remains intact even if the notebook server is terminated.

JupyterHub provides flexibility to add multiple persistent volumes to notebooks. This functionality comes in handy when users need a shared storage to exchange code and artifacts.

There are a few common usage patterns when providing storage to notebook servers:

- Block storage – Notebook servers get a persistent volume that's backed up a block storage service like Amazon EBS. Block storage offers high throughput and low latency storage, but it can be expensive as you must provision one fixed size volume per user. For example, every Jupyter user gets a 10 Gi volume in your current configuration.
- Shared storage – Notebook servers store data using a persistent volume that's backed by a shared storage service like NFS or Amazon EFS. Shared storage is more cost effective than block storage, but doesn't offer the same performance as block storage.
- Block and shared storage – Notebook servers have multiple persistent volumes, one backed by block storage to store user's workspace data and another shared volume that is used as a repository for shared code and artifacts.

If you're unfamiliar with different storage options, AWS documentation has a helpful page to explain the difference between block, object, and file storage uses: <https://aws.amazon.com/compare/the-difference-between-block-file-object-storage/>.

Besides block storage, all notebook servers have access to shared storage. The shared EFS filesystem is mounted at `/home/shared`. Whenever users need to exchange data, they can do so by using the shared directory. To test this shared storage functionality, you can create a test file in `/home/shared` as "mluser". In a new browser session (using private or incognito modes), login to JupyterHub as the "mladmin" user and you'll be able to see the files you created as "mluser" at `/home/shared`.

3.4.4 Customizing user environment

The notebook server that got created for the user "mluser" is currently running on the only node in your cluster. Since this node doesn't come with a graphics processing unit (GPU), your notebook server doesn't have GPU resources. However, you have the flexibility to add GPU support to your notebook servers if needed.

Kubernetes allows you to create nodes with GPUs and instruct it to run notebook servers on nodes that have one or more GPUs attached. This way, users who require GPU acceleration for their workloads, such as training deep learning models or running computationally intensive simulations, can leverage the power of GPUs seamlessly.

It's important to note that GPU resources can be expensive, and not every user might require them. If you provision GPU nodes for all users, you might end up paying for expensive resources that go underutilized. To optimize costs, you can adopt a more targeted approach.

JupyterHub allows you to create custom profiles for notebook servers where you can specify the CPU, GPU, and memory requirements, runtimes, and startup scripts for notebook servers.

Consider that you need three notebook profiles:

- All Spark notebook – For data scientists that use Python, Scala, and R
- Python only – For data science users that only use Python
- TensorFlow – For users that will create models using TensorFlow and need a GPU

You can specify these profiles in the `jupyterhub-values.yaml` file as shown in listing 3.2.

Listing 3.2 Snippet of `jupyterhub-values.yaml` with multiple profiles

```
singleuser:  
  profileList:  
    - display_name: "All Spark environment" #A  
      description: "Python, Scala, R and Spark Jupyter Notebook  
      Stack"  
        default: true #B  
        kubespawner_override:  
          image: jupyter/all-spark-notebook #C  
    - display_name: "Python only"  
      description: "Data Science Jupyter Notebook Python Stack"  
      kubespawner_override:  
        image: jupyter/datascience-notebook  
    - display_name: "TensorFlow with GPU"  
      description: "Jupyter Notebook Python Stack with  
      TensorFlow"  
        kubespawner_override:  
          image: jupyter/tensorflow-notebook  
          extra_resource_guarantees:  
            nvidia.com/gpu: "1"  
          extra_resource_limits:  
            nvidia.com/gpu: "1"  
#A The name to be displayed to users when they are selecting a Jupyter environment  
#B This boolean flag indicates that this profile is set as the default option  
#C The container image to be used for this Jupyter environment
```

Each profile uses a different container image provided by Jupyter Docker Stacks (<https://jupyter-docker-stacks.readthedocs.io/en/latest/index.html>), which are a set of container images created by the Jupyter community.

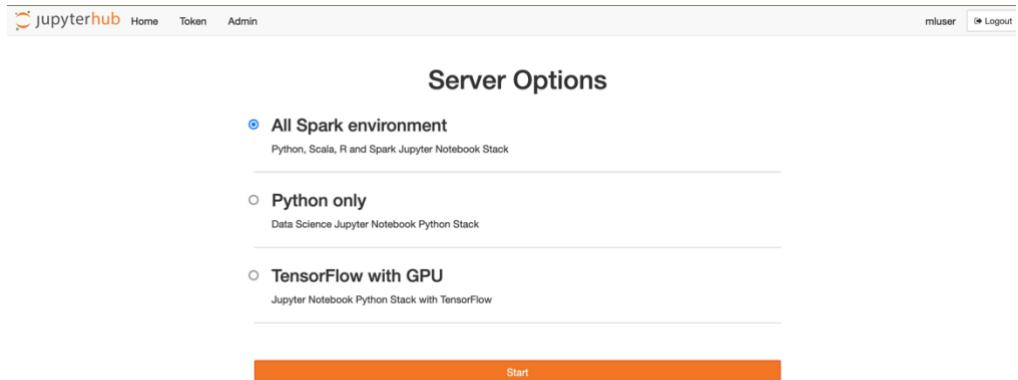


Figure 3.12 Screenshot of JupyterHub with notebook profile. Each profile offers a customized development environment that's designed to match the needs of a project.

NOTE While using container images from public registries is acceptable for testing and debugging purposes, avoid deploying images from untrusted registries in production. It is a best practice to host container images in a private, secure container registry. To improve security, scan container images for vulnerabilities, malware, and policy violations before making them available for production usage.

3.4.5 Enabling GPU workloads in Kubernetes

Out of the box, your Kubernetes cluster is not ready to run GPU workloads. If you attempt to start a “TensorFlow with GPU” notebook, the server will fail to start. The notebook server fails to start because it requires a GPU and the cluster currently doesn’t have any node with the GPU.

In fact, there are two problems. The first is indeed the fact that currently no worker nodes have a GPU. But, even if one of the workers node did have one, Kubernetes will have no awareness of the availability of this resource.

You see by default, when a worker nodes start, the kubelet running on the worker node gathers hardware information about the node. It collects data points such as number of CPU cores available, total memory, available disk space. What it doesn’t collect by is the number of GPUs. So, when a Pod requests a GPU, Kubernetes has no idea how to schedule that Pod.

For Kubernetes to know when a worker node has a GPU installed, we must install the NVIDIA Device Plugin. This plugin is a Daemonset that runs on each node in the cluster and is responsible for discovering and reporting the presence of GPUs to the Kubernetes scheduler. Once installed, the plugin exposes GPUs as

a schedulable resource, allowing Kubernetes to allocate and manage GPU resources across the cluster.

Just like you need to install drivers to use NVIDIA GPUs, you must install the drivers and plugins on worker nodes before Kubernetes can run GPU workloads. Thankfully, NVIDIA makes it easy to prepare worker nodes with GPUs in a Kubernetes environment. The *NVIDIA GPU Operator* (<https://github.com/NVIDIA/gpu-operator>) is a Kubernetes operator that simplifies the installation of NVIDIA GPU drivers and configuration.

NOTE In the next steps, we'll enable GPU support in the cluster. If you are in a cloud environment, this will enable your cluster to create worker nodes with GPUs. EC2 instances with GPU are quite expensive. So before you enable this support, ensure that you're okay with the cost of creating cloud instances. To provide an estimate, in AWS, running a small GPU instance like g4.xlarge costs roughly fifty cents an hour as of writing.

Install NVIDIA GPU Operator using Helm:

```
$ helm repo add nvidia https://helm.ngc.nvidia.com/nvidia
$ helm repo update
$ helm upgrade -i gpuo nvidia/gpu-operator \
  -n kube-system \
  --values nvidia-gpu-operator.yaml
```

The NVIDIA GPU Operator plays a crucial role in managing GPU resources. Among other things, it runs the GPU Feature Discovery (<https://github.com/NVIDIA/gpu-feature-discovery>) tool, which detects GPUs on a node and advertises their availability to Kubernetes using node labels. This labeling mechanism enables Kubernetes to identify and allocate GPU resources to workloads.

When you attempt to start a "TensorFlow with GPU" notebook server in JupyterHub, the corresponding Pod will initially remain in the "Pending" state. This is because none of the existing nodes in the cluster has a GPU available. To address this situation, we must instruct Amazon EKS that it can provision EC2 instances with GPUs. This is done by creating a new node pool. Once you create this node pool, Karpenter will automatically provision a new node with a GPU. Listing 3.3 includes a Karpenter node pool definition.

Listing 3.3 Karpenter GPU node pool

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: gpu #A
```

```

spec:
  template:
    spec:
      nodeClassRef:
        group: eks.amazonaws.com
        kind: NodeClass
        name: default
      requirements:
        - key: "eks.amazonaws.com/instance-category"
          operator: In
          values: ["g", "p"] #B
        - key: "kubernetes.io/arch"
          operator: In
          values: ["amd64"] #C
      taints:
        - key: nvidia.com/gpu #D
          effect: NoSchedule
      limits:
        cpu: 1000
      disruption:
        consolidationPolicy: WhenEmpty
        consolidateAfter: 60s
      weight: 20
#A Name of the node pool
#B This node pool creates P or G instance types (these Amazon
EC2 instance families have one or more GPUs)
#C Worker nodes must not be ARM-based
#D All worker nodes will have a taint by default ensuring only
Pods that require a GPU get scheduled on nodes with GPUs

```

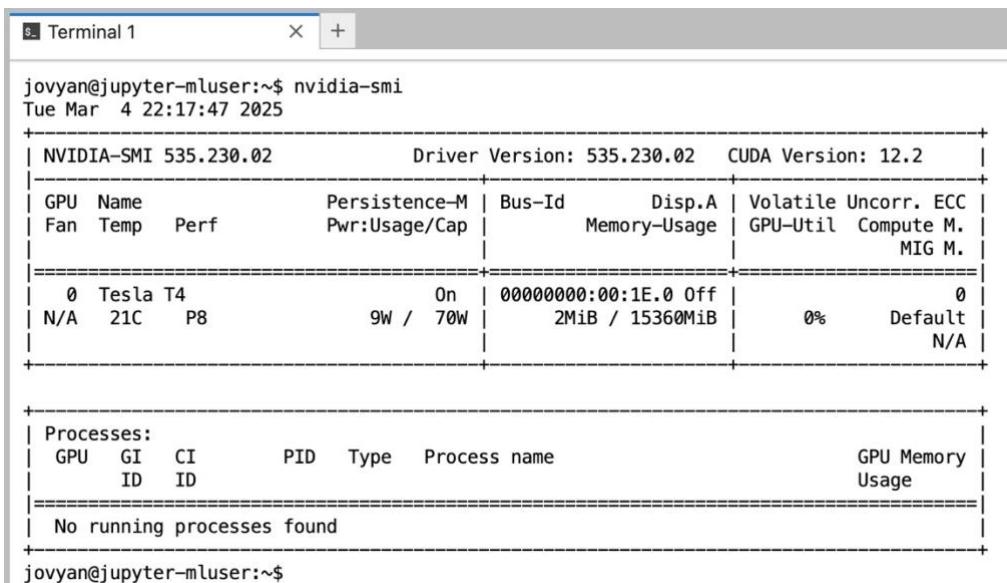
The process of creating a new node with a GPU and installing the necessary NVIDIA utilities can take up to five minutes. During this time, the following steps occur:

13. Node Provisioning – Karpenter initiates the creation of a new node in the cluster, ensuring that it meets the GPU requirements specified by the Pod.
14. GPU Driver Installation – Once the new node is provisioned, the NVIDIA GPU Operator installs the appropriate GPU drivers and utilities on the node.
15. GPU Feature Discovery – The GPU Feature Discovery tool runs on the new node, detecting the available GPUs and advertising their capabilities using Kubernetes node labels.
16. Pod Scheduling – With the GPU resources now available and labeled, Kubernetes can schedule the "TensorFlow with GPU" notebook server Pod on the newly provisioned node.

To accommodate the potentially longer startup time required for provisioning a new node with a GPU and setting up the necessary software, we have increased

the notebook server startup timeout to ten minutes (using the Helm values file). This configuration ensures that JupyterHub does not prematurely terminate the notebook server startup process, allowing sufficient time for the GPU resources to be made available and the Pod to be scheduled successfully.

```
singleuser:  
  startTimeout: 600
```



The screenshot shows a terminal window titled "Terminal 1". The command "nvidia-smi" is run, displaying GPU information. The output includes:

```
jovyan@jupyter-mluser:~$ nvidia-smi  
Tue Mar  4 22:17:47 2025  
+-----+  
| NVIDIA-SMI 535.230.02      Driver Version: 535.230.02    CUDA Version: 12.2 |  
+-----+  
| GPU  Name        Persistence-M | Bus-Id     Disp.A  | Volatile Uncorr. ECC  | | | | |
| Fan  Temp     Perf  Pwr:Usage/Cap | Memory-Usage | GPU-Util  Compute M.  |  
|          |          |          |          |          |          | MIG M. |  
+-----+  
| 0  Tesla T4           On  00000000:00:1E.0 Off |          0 | | | | | |
| N/A  21C   P8          9W / 70W | 2MiB / 15360MiB | 0%      Default |  
|          |          |          |          |          |          | N/A  |  
+-----+  
  
+-----+  
| Processes:          GPU Memory  |  
| GPU  GI  CI    PID  Type  Process name | Usage  |  
| ID   ID          |  
+-----+  
| No running processes found |  
+-----+  
jovyan@jupyter-mluser:~$
```

Figure 3.13 NVIDIA GPU Operator configures a worker node with GPU to support Kubernetes workloads. Once installed, you can start the “TensorFlow with GPU” notebook server and list the attached GPU by running the nvidia-smi command.

By leveraging the NVIDIA GPU Operator and Karpenter’s autoscaling capabilities, your Kubernetes cluster can dynamically provision GPU resources on-demand. This approach ensures efficient resource utilization and enables seamless access to GPU-accelerated workloads, such as those required for deep learning or scientific computing tasks.

3.4.6 Reducing wastage by shutting down idle notebooks

Cloud environments, whether public or private, offer elastic compute capacity, allowing workloads to seamlessly scale up and down based on demand. This scalability is particularly beneficial in scenarios like the JupyterHub environment

deployed in your Kubernetes cluster. Initially, with only one user, JupyterHub creates a single Pod to host the user's notebook server. However, as more users onboard, JupyterHub dynamically creates additional Pods to accommodate their notebook servers.

While this on-demand provisioning of resources is advantageous, it can also lead to inefficiencies if not managed properly. Once a user's notebook server is provisioned, it continues to consume compute resources, even when the user is not actively utilizing the server or has completed their work. This idle resource consumption can result in wasted resources and unnecessary costs.

To address this challenge, JupyterHub provides a feature that allows you to stop a user's notebook server after it has been idle for a specified period. By implementing this idle server timeout, you can effectively optimize resource utilization and reduce waste.

In `jupyterhub-values.yaml`, we have enabled culling user notebooks after they have been idle for more than 3600 seconds. This configuration is specified as follows:

```
cull:  
  enabled: true  
  timeout: 3600
```

Additionally, we have set the maximum age of any notebook server to not exceed one week (604800 seconds). This configuration ensures that running notebook servers will be terminated after a week, regardless of their activity status:

```
cull:  
  maxAge: 604800
```

NOTE every user's notebook server in JupyterHub has an associated persistent volume. This persistent volume ensures that user data remains intact even when their notebook server is terminated. When JupyterHub terminates a user's notebook server, whether due to inactivity, resource constraints, or other reasons, the persistent volume containing the user's data is not affected. The data remains safely stored and available for future use. Subsequently, when the user logs in again after their notebook server has been terminated, JupyterHub will create a new Pod for their notebook server. However, this new Pod will have the same persistent volume attached, allowing the user to seamlessly pick up where they left off.

Setting maximum age helps to ensure that the containers used for notebook servers run the most up-to-date versions of JupyterHub and included libraries, thereby improving the security and reliability of the notebook servers.

By regularly terminating and recreating notebook servers, you ensure that the latest security patches and updates are applied, reducing the risk of vulnerabilities in outdated software versions. Users also benefit from a consistent and up-to-date environment, reducing the likelihood of encountering issues caused by outdated software or configurations.

As JupyterHub terminates idle notebook server Pods, Karpenter will terminate any unused nodes automatically. This dynamic resource management capability helps optimize resource utilization and reduce costs by ensuring that resources are allocated only when needed.

3.4.7 Optimizing GPU node utilization

GPUs are costly resources, and their efficient utilization is crucial for cost-effective operations. By default, Kubernetes does not differentiate between GPU and non-GPU workloads when scheduling, which can lead to suboptimal use of GPU nodes. To ensure that only workloads requiring GPUs are scheduled on GPU nodes, you can leverage Kubernetes' taints and tolerations mechanism.

Taints are applied to nodes to mark them as having special properties that should not accept any pods unless those pods explicitly tolerate the taints. In the Karpenter NodePool configuration, we have added a taint to nodes with GPU:

```
taints:
  - key: nvidia.com/gpu
    effect: NoSchedule
```

With this configuration, Karpenter adds a taint to the nodes in the "gpu" node pool, indicating to Kubernetes that no pods should be scheduled on these nodes unless they tolerate the taint.

Tolerations are applied to Pods to allow them to be scheduled on nodes with specific taints. To ensure that only GPU workloads (for example the "TensorFlow with GPU" notebook server) are scheduled on GPU nodes, a taint is added to any GPU Pods. For example, in the "jupyter-values.yaml" file, there's a toleration added to "TensorFlow with GPU" notebook profile as shown in listing 3.4.

Listing 3.4 Snippet of JupyterHub configuration specifying tolerations for GPU pods

```
- display_name: "TensorFlow with GPU"
  description: "Jupyter Notebook Python Stack with TensorFlow"
  kubespawner_override:
    image: jupyter/tensorflow-notebook
    extra_resource_guarantees:
      nvidia.com/gpu: "1"
  extra_resource_limits:
    nvidia.com/gpu: "1"
```

```

tolerations: #A
  - key: nvidia.com/gpu #B
    operator: Exists
    effect: NoSchedule
#A Tolerations added to any Pods that require a GPU
#B Key of the taint

```

By utilizing taints and tolerations, Karpenter ensures that GPU nodes are terminated when there are no Pods in the cluster requiring GPU resources. Without this mechanism, Kubernetes could potentially schedule non-GPU Pods on GPU nodes. This would prevent Karpenter from shutting down those nodes when they are not needed for GPU workloads.

To avoid such scenarios, the taints and tolerations ensure that non-GPU workloads are only scheduled on nodes without a GPU. This way, GPU nodes remain dedicated to GPU-intensive workloads, and Karpenter can efficiently scale down the GPU node pool when the demand for GPU resources decreases.

3.5 Summary

- Amazon EKS provides a production-grade Kubernetes cluster making it easier for you create Kubernetes clusters in the cloud.
- Karpenter is a Kubernetes autoscaler that dynamically scales worker nodes when cluster runs out of capacity or is over-provisioned.
- AWS Load Balancer is a Kubernetes controller that allows you to provision load balancers using Kubernetes.
- The NGINX Ingress Controller is a cloud-agnostic implementation of Kubernetes Ingress Controller for NGINX that can load balance Websocket, gRPC, TCP and UDP applications. It allows you to expose multiple Kubernetes service using a single load balancer.
- Keycloak is a Kubernetes-compatible open-source identity provider for securing access to applications.
- JupyterHub on Kubernetes creates scalable, customizable, and optimized notebook servers for users. It supports terminating inactive notebook servers automatically for resource efficiency and cost optimization.
- NVIDIA GPU Operator simplifies installing drivers and configuring GPUs in a Kubernetes environment.
- Using taints and tolerations, you can ensure that GPU resources are used effectively and nodes with GPUs are only created with there's a workload that requires a GPU.

4

Scaling Data Pipelines with Kubernetes

This chapter covers

- Understanding data pipelines
- Transitioning data engineering from a notebook to a pipeline
- Running data pipelines with Apache Airflow
- Distributed data processing pipelines using Apache Spark

In the previous chapter, you learned how to create a scalable Jupyter notebook environment on Kubernetes. Combining JupyterHub with Keycloak, we empowered users to create personalized notebook servers tailored to their specific needs, while ensuring secure access and data persistence.

In this chapter, we will explore how Kubernetes helps you scale data pipelines. As you learned in Chapter 1, the efficacy of an ML model depends on the quality of data used to train the model. To prepare data for model training, data scientists perform a series of data processing steps to collect, cleanse, and structure data. This process is commonly known as *data engineering*.

Data pipelines are crucial in data engineering, serving as the backbone for collecting, processing, and delivering data for analysis and model training. They automate ingesting data from various sources, such as databases, APIs, streaming data, and files. They handle the extraction and integration of structured, unstructured, or semi-structured data from disparate sources into a centralized location for further processing.

One key problem with raw data is that it is rarely in a format suitable for machine learning. Therefore, you often need to clean, transform, and enrich data. Data pipelines incorporate processes like data validation, filtering, deduplication, and formatting to ensure data quality and consistency. They periodically ingest data from external systems into data stores.

Let's explore this process using an example.

4.1 Understanding the basics of data processing

Let's get our hands dirty and review the process of preparing data at a conceptual level.

First, to download the dataset, you'll need a Kaggle (Kaggle.com) account. After logging in, navigate to your Kaggle account settings and create an API token. The API token contains your Kaggle username and secret key, which allow you to download datasets programmatically.

Once you have your Kaggle API key, start a Jupyter notebook (use "All Spark environment" server option). Since the Jupyter container image does not include Kaggle libraries, you will need to install it using pip. You can run shell commands from a notebook cell by adding an exclamation mark ("!"). For example, to install the kaggle command line utility (or CLI) using pip, you'll run '!pip install kaggle'. Kaggle also provides Python API but we're going to ignore it for the purposes of this discussion.

Listing 4.1 shows the code to configure Kaggle and download a dataset. The dataset contains raw data on medical conditions and associated drug treatments. Why did we choose this example dataset? Primarily because its small enough to serve as an example for this exercise. The specific contents of the dataset themselves are unimportant to us. The purpose of this exercise is to illustrate how raw data goes through a series of steps before it is ready for consumption.

Listing 4.1 Downloading datasets from Kaggle

```
!pip install kaggle
import os

os.environ['KAGGLE_USERNAME'] = "YOUR KAGGLE USERNAME" #A
os.environ['KAGGLE_KEY'] = "YOUR KAGGLE KEY" #B

import kaggle
kaggle.api.authenticate()

!kaggle datasets download \ #C
    -d himalayaashish/medi-drug-dataset \
    --unzip \
    -p /home/shared/
#A Your Kaggle username
#B Your Kaggle API key
#C Use kaggle CLI to download dataset
```

```
[3]: !pip install kaggle
Collecting kaggle
  Using cached kaggle-1.6.14-py3-none-any.whl
Requirement already satisfied: six<=1.10 in /opt/conda/lib/python3.11/site-packages (from kaggle) (1.16.0)
Requirement already satisfied: certifi>=2023.7.22 in /opt/conda/lib/python3.11/site-packages (from kaggle) (2023.7.22)
Requirement already satisfied: python-dateutil in /opt/conda/lib/python3.11/site-packages (from kaggle) (2.8.2)
Requirement already satisfied: requests in /opt/conda/lib/python3.11/site-packages (from kaggle) (2.31.0)
Requirement already satisfied: toml in /opt/conda/lib/python3.11/site-packages (from kaggle) (4.66.1)
Collecting python-slugify (from kaggle)
  Using cached python_slugify-8.0.4-py2.py3-none-any.whl.metadata (8.5 kB)
Requirement already satisfied: urllib3 in /opt/conda/lib/python3.11/site-packages (from kaggle) (2.0.7)
Requirement already satisfied: bleach in /opt/conda/lib/python3.11/site-packages (from kaggle) (6.1.0)
Requirement already satisfied: webencodings in /opt/conda/lib/python3.11/site-packages (from bleach->kaggle) (0.5.1)
Collecting text-unidecode>=1.3 (from python-slugify->kaggle)
  Using cached text_unidecode-1.3-py2.py3-none-any.whl.metadata (2.4 kB)
Requirement already satisfied: charset-normalizer<4,>=2 in /opt/conda/lib/python3.11/site-packages (from req
[4]: import os
os.environ['KAGGLE_USERNAME'] = " "
os.environ['KAGGLE_KEY'] = "1"
[5]: import kaggle
kaggle.api.authenticate()
Download the dataset
[10]: !kaggle datasets download -d himalayaashish/medi-drug-dataset --unzip -p /home/shared/
Dataset URL: https://www.kaggle.com/datasets/himalayaashish/medi-drug-dataset
License(s): unknown
Downloading medi-drug-dataset.zip to /home/shared
  0%|          | 0.00/57.8k [00:00<?, ?B/s]
 100%|██████████| 57.8k/57.8k [00:00<00:00, 1.38MB/s]
```

Figure 4.1 Datasets can be downloaded from Kaggle using cli or API.

This process of downloading data from a source system is called *data ingestion*. With the dataset available locally, we can start data preparation and perform *exploratory data analysis* (EDA). EDA is an approach to analyzing datasets by summarizing their key characteristics, typically using data visualization methods. “EDA helps determine how best to manipulate data sources to get the answers you need, making it easier for data scientists to discover patterns, spot anomalies, test a hypothesis, or check assumptions”, according to IBM (<https://www.ibm.com/think/topics/exploratory-data-analysis>).

The primary goal of EDA is to understand the data and identify anomalies, missing values, and outliers. When using Python for data analytics, data scientists prefer Python libraries like Pandas, Numpy, and Matplotlib for data analysis. Since the JupyterHub container image for this notebook server already includes these libraries, we do not need to install them using pip. Listing 4.2 shows the steps to load the data from the CSV file into a Pandas DataFrame and display the first five rows of the DataFrame. Following steps are also available as a Jupyter Notebook on this book’s GitHub repository at “Chapter 4/notebook/medidata.ipynb”.

Listing 4.2 Loading the contents of the CSV file into a pandas DataFrame

```
import pandas as pd
```

```
df = pd.read_csv('/home/shared/Medi_Drug.csv') #A  
df.head() #B
```

#A Load the CSV file into a pandas DataFrame

#B Show the first five rows of the DataFrame

`df.head()` displays the DataFrame's columns and a sample of their contents. Pandas infers column names from the first row of the CSV file.

df.head()									
	Condition	Drug	Indication	Type	Reviews	Effective	EaseOfUse	Satisfaction	Information
0	Acute Bacterial Sinusitis	Levofloxacin	On Label	RX	994 Reviews	2.52	3.01	1.84	\\n t t t t t tLevofloxacin is used to treat a ...
1	Acute Bacterial Sinusitis	Levofloxacin	On Label	RX	994 Reviews	2.52	3.01	1.84	\\n t t t t t tLevofloxacin is used to treat a ...
2	Acute Bacterial Sinusitis	Moxifloxacin	On Label	RX	755 Reviews	2.78	3.00	2.08	\\n t t t t t t This is a generic drug. The ave...
3	Acute Bacterial Sinusitis	Azithromycin	On Label	RX	584 Reviews	3.21	4.01	2.57	\\n t t t t t tAzithromycin is an antibiotic (m...
4	Acute Bacterial Sinusitis	Azithromycin	On Label	RX	584 Reviews	3.21	4.01	2.57	\\n t t t t t tAzithromycin is an antibiotic (m...

df.columns									
Index(['Condition', 'Drug', 'Indication', 'Type', 'Reviews', 'Effective', 'EaseOfUse', 'Satisfaction', 'Information'], dtype='object')									

Figure 4.2 The `df.head()` function displays the first five rows of a DataFrame. You can view the column names using `df.columns`.

Sometimes the raw data may contain data irrelevant to your use case. In such cases, it is customary to remove unnecessary information. For example, after inspecting the data, we may conclude that the “Information” column is “noise”, and we don’t need to retain it. We can delete the “Information” column using:

```
df.drop(columns='Information', inplace=True)
```

Another common task that’s performed during data preparation is *data cleansing*. To cleanse the dataset, you may have to drop observations that have missing values. Another approach is to impute or fill in the missing values. For example, the “Indication” column contains whether a drug was prescribed as approved by the Food & Drug Administration (FDA). Some observations contain junk data in the “Indication” attribute.

```
[32]: df['Indication'].value_counts()
```

Indication	Count
On Label	1723
Off Label	465
\r\n	31

```
[32]: Indication
      On Label    1723
      Off Label   465
      \r\n          31
      Name: count, dtype: int64
```

Figure 4.3 Summarize the occurrences of each unique value in the “Indication” column of the DataFrame to find errors or missing data.

It is a best practice to fill missing or invalid values in the dataset. For example, we can perform a string replacement operation on the “Indication” column to replace “\r\n” with “Unidentified”:

```
Replace \r\n with "Unidentified"
[34]: df['Indication'] = df['Indication'].str.replace('\r\n', 'Unidentified')
df['Indication'].value_counts()
```

Indication	Count
On Label	1723
Off Label	465
Unidentified	31

```
[34]: Indication
      On Label    1723
      Off Label   465
      Unidentified 31
      Name: count, dtype: int64
```

Figure 4.4 Data cleansing involves replacing missing or invalid values.

Let's pause here and review what we have accomplished so far. Our notebook now contains the steps to download the dataset and cleanse it. By sharing this notebook with your colleagues, they can rerun the cells to reproduce the process of downloading and processing the data. What you've essentially built is a naïve data pipeline.

Keep in mind that production pipelines have many more steps than we've shown in the example. These steps may include additional ML specific tasks like:

- Vectorization – Converting strings, images, and other non-numerical attributes into numerical vectors
- Normalization – Rescaling features to a common range
- Bucketing/Windowing – Grouping timeseries or categorical data into discrete intervals or categories

We'll stop our data processing steps at this point because the intent of this chapter is to illustrate how to run data pipelines on Kubernetes. We must not deviate too much into the intricacies of data science. If you're interested in learning more about data science and engineering, we recommend Andrew Treadway's *Software Engineering for Data Scientists* (Manning 2024).

Before we proceed, store the processed data in a CSV file:

```
df.to_csv('/home/shared/notebook-output.csv', index=False)
```

4.2 Data processing in pipelines

The reason we called the notebook you created in section 4.1 a naïve implementation of a data pipeline is because while it contains the steps you'd perform for data preparation, it isn't automated, scalable, or robust. Jupyter notebooks are excellent tools for data scientists to perform exploratory data analysis, prototyping, and ad hoc data processing tasks, but they are not well-suited for production-ready data pipelines.

Keep in mind that data ingestion and processing are not a one-time exercise. Real-world systems are constantly generating new data. To maintain the freshness of the input data for machine learning models, you must refresh or add new data regularly.

Data pipelines encapsulate the series of steps needed to process data. Data pipelines are workflows that are designed to be automated and reproducible. They not only save time but also ensure that data processing is consistent and error-free.

Another reason you cannot use notebooks for data preparation is that a notebook has limited CPU, memory, and disk resources. When dealing with larger datasets, you may not be able to manipulate data without running into resource contention. Data pipelines can be designed to handle this complexity. They can leverage distributed data analytics tools like Apache Spark to process large volumes of data.

Data pipelines also make data processing steps more robust, as they are usually orchestrated using *workflow management tools*. These tools enable data pipelines to incorporate monitoring, retries, and error handling mechanisms, allowing you to track the progress of data processing individual tasks. To contrast, if you were to perform data processing in a single script, and the script fails midway, you would have to rerun the entire script after fixing errors. There's no easy way to automatically skip the tasks that didn't fail. Workflow management tools are designed to handle such failures. They include features that intelligently resume failed tasks without rerunning them entirely.

When building an ML system, data scientists usually start by exploring, cleansing, and processing the dataset in a notebook. Once data processing steps produce desired results, data scientists hand over the code to the data engineering team. It then falls upon the data engineering team to convert the

notebook into a modular data pipeline that can be orchestrated by a workflow management tool like Apache Airflow.

This conversion process involves transforming each data processing step into a modular component, emphasizing reusability and flexibility. Designing each pipeline task as a self-contained module improves its maintainability and reusability. This modular approach allows for easier updates, testing, and integration of new data sources or processing steps, ultimately leading to a more flexible and scalable ML system.

Please note that Airflow is one of the many workflow orchestration tools geared towards addressing the challenges of data pipelines. There are many other open source and commercial alternatives to Airflow. We chose Airflow in this book because it has the most widespread adoption. Moreover, almost all workflow management tools are conceptually similar. If you learn Airflow and its concepts, you'll be well-equipped to work with other orchestration tools in the future.

The principles we'll explore, such as task modularity, dependency management, and pipeline scalability, are transferable across different platforms. Whether you end up using Airflow or any other tool in your professional work, the fundamental ideas of building robust, scalable data pipelines remain consistent.

As we progress through this chapter, we'll transform our notebook-based data processing into a data pipeline orchestrated by Apache Airflow.

4.3 Introducing Apache Airflow

Apache Airflow is an open-source platform for programmatically creating, scheduling, and monitoring workflows. A *workflow* in Airflow consists of a sequence of tasks. Airflow is widely used by data professionals to manage complex data pipelines and workflows, particularly in cloud-native environments. Airflow is not a data processing tool. Its job is to run a collection of data processing tasks periodically, automating processes such as data ingestion, processing, cleansing, and structuring.

Consider the case of building a model that predicts stock prices. To build such a model, you'd need historical data for stock prices. But not just that, you'd also need to download fresh data at the end of every trading day to maintain the accuracy of the model. You can write a script that fetches this data every night. But the question is, where do you run it?

The simplest way to do this would be to write a script that downloads data and schedule it using `cron`, which is a command line utility for job scheduling in Unix-like operating systems. However, using `cron` for complex workflows has limitations. Cron jobs are difficult to monitor, lack a user-friendly interface, and

do not handle dependencies between tasks well. This is where Apache Airflow excels. Airflow allows you to define workflows as *Directed Acyclic Graphs (DAGs)*, where each node represents a task, and edges define dependencies between these tasks. This structure makes it easy to visualize and manage complex workflows.

For instance, in the stock price prediction example, you can create a DAG that includes tasks for downloading daily stock data, cleaning and processing it, training the machine learning model, and finally making predictions. Each of these tasks can be scheduled to run at specific times or triggered by external events. Airflow ensures that tasks are executed in the correct order, respecting all dependencies.

Airflow is a versatile tool. It supports a wide range of operators, which are predefined tasks that perform operations, such as running a Python function, a database query, a Bash script, or interacting with cloud services like AWS, Google Cloud, and Azure. This extensibility allows you to integrate Airflow with virtually any system or service, making it an ideal tool for managing data pipelines and more.

A DAG for grocery shopping

To better understand the concept of DAGs, let's consider a relatable example - grocery shopping.

Grocery shopping can be defined as a workflow with several distinct steps or tasks. Such a workflow might include:

1. Drive to the grocery store
2. Find the desired items and place them in cart
3. Pay for the items
4. Return home

Now, imagine representing this workflow as a DAG. Each step becomes a node or task in the graph, and the dependencies between these tasks are represented by edges or arrows.

Drive to Store > Buy Items > Pay for Items > Return Home

The arrows indicate that a task can only start after the previous task has completed successfully. For instance, you can't pay for your items until you've selected and collected them from the store shelves.

Apache Airflow supports workflows with sequential as well as parallel steps. This flexibility allows users to design complex data pipelines that can efficiently handle various tasks, whether they need to be executed one after another or simultaneously.

This directed flow, where tasks have an explicit order and dependencies, is what makes it an “acyclic” graph - there are no circular dependencies, meaning a task can never depend on itself or a task that ultimately depends on it. In a DAG, information only flows forward, and never backwards.

In Airflow, you define these workflows as DAGs in Python. Each DAG consists of one or more tasks, dependencies between them, and scheduling intervals that determine when the workflow should run.

For example, we can convert the steps we performed in section 4.1 into individual tasks. Such a DAG will have one task for each of these steps:

- Download the dataset from Kaggle
- Remove unneeded columns
- Replace missing data
- Save the processed file to shared storage

By representing your data pipelines as DAGs, Airflow ensures that tasks are executed in the correct order, handling dependencies and failures appropriately. You can visualize the entire workflow, monitor its progress, and troubleshoot issues.

By leveraging Airflow's DAG structure, you can define dependencies and orchestrate tasks in a way that optimizes resource utilization and reduces overall execution time. For instance, tasks that are independent of each other can be configured to run in parallel, thereby speeding up the workflow and improving performance. This capability is particularly useful in scenarios where large volumes of data need to be processed quickly, such as in ETL (Extract, Transform, Load) operations, data analysis, and machine learning model training.

The key advantage of using DAGs in Airflow is that they provide a structured and maintainable way to define and orchestrate complex workflows, making it easier to manage and scale your data pipelines as your requirements grow.

4.4 *Installing Airflow*

By now, you should have become familiar with the way we deploy applications in Kubernetes. Just as we did with NGINX, Keycloak, and JupyterHub, we'll use Helm to install Airflow. To customize the setup, we have created a values file. This values file includes configuration that integrates Airflow with Keycloak for

user authentication. We've also configured Airflow to import DAGs from a shared storage. This shared storage is also attached to Jupyter notebooks. To submit DAGs to Airflow, you write a DAG file and copy it to the shared directory, which is mounted at /home/shared in notebook servers. But before you install Airflow, let's setup Keycloak so we can login to the Airflow UI using single sign-on.

4.4.1 Configuring Keycloak to authenticate Airflow users

Just like JupyterHub, Airflow provides several ways to restrict access to the webserver. We'll use Keycloak to authenticate users before they can access the Airflow UI. Let's recap the process of setting up clients in Keycloak:

- 17.Create a client in Keycloak and provide the client application's URLs
- 18.Create a client secret
- 19.Configure the client to use OAuth authentication

You'll repeat these steps several times throughout this book.

To demonstrate the role-based access control capabilities of Airflow and Keycloak, this time we'll also create groups. We'll create two groups: admin and public. Admin users have full control over the Airflow cluster, whereas users in the public group will have read only access.

Instead of creating a client manually, this time you'll import a client. Keycloak allows you to export client configuration. We've included the export of Airflow client we created in the book's Git repository at Chapter 4/airflow/keycloak-airflow-client.json.

Please run the following command to change the values of client URLs in the file to match your deployment:

```
$ envsubst < keycloak-airflow-client.json > keycloak-airflow-client.modified.json
```

To import the client, login to Keycloak and choose Clients > Import client and select keycloak-airflow-client.modified.json. Leave other settings to their default values.

After you've imported the file and created a new client, go to the Clients > Roles tab and create two client roles: airflow_admin and airflow_public.

Next, go to Users > mluser > Role mapping and assign the mluser user the airflow_admin client role as shown in figure 4.5. This role gives mluser Airflow administrative privileges.

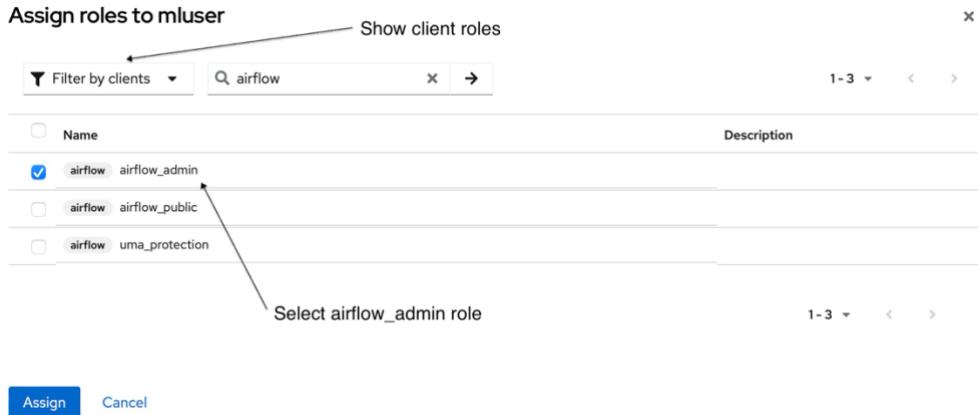


Figure 4.5 Assign airflow_admin role to mluser. This role will give Airflow administrative privileges to the user.

4.4.2 Deploying Airflow using Helm

With Keycloak setup for Airflow user authentication, it's time to deploy. We've created a values file to configure Airflow located at Chapter 4/airflow/airflow-values.yaml.template. This YAML file changes several default values:

- It sets KubernetesExecutor as the default Airflow executor
- It creates a shared persistent volume (using Amazon EFS) that's used to share DAGs with Airflow
- Airflow components write logs to the shared volume
- Airflow UI is accessible to users through an Ingress
- The Airflow webserver uses Keycloak to authenticate users

Create a namespace for Airflow:

```
$ kubectl create namespace airflow
```

Make sure that the \$DOMAIN environment variable is set to your domain (if not, set it by running `export DOMAIN=<YOUR DOMAIN>`). Then generate a Helm values file to match your environment.

```
$ envsubst < airflow-values.yaml.template > airflow-values.yaml
```

We've configured Airflow to use the NFS-based filesystem we created in the previous chapter for storing logs and DAGs. This shared persistent volume allows users to create DAGs using Jupyter notebooks and submit them to Airflow. To create a `PersistentVolumeClaim` (PVC) and a `PersistentVolume` (PV) that points to the shared EFS filesystem, run:

```
$ sh create-efs-pv.sh
```

If you aren't operating in AWS, please create a PVC for Airflow. This PVC should point to the same NFS-based filesystem as the one used for JupyterHub.

Next, install Airflow using custom Helm chart values:

```
$ helm repo add apache-airflow https://airflow.apache.org
$ helm repo update
$ helm upgrade -i "airflow" \
    apache-airflow/airflow \
    --namespace airflow \
    --values ./airflow-values.yaml
```

Once Helm completes the installation, you'll have Airflow components running along with a Postgres database, which is used to store Airflow metadata. You can then login to Airflow as 'mluser' by going to [https://platform.\\${DOMAIN}/airflow](https://platform.${DOMAIN}/airflow).

When users login, Airflow gives them permissions based on their groups. For example, since the user 'mluser' is part of the 'airflow_admin' group, the user will have administrative privileges. To customize role assignment, you can configure the 'webserverConfig' key in the `airflow-values.yaml` file. To learn more about Airflow webserver configuration, please see: <https://airflow.apache.org/docs/apache-airflow/stable/configurations-ref.html>

4.4.3 Exploring Airflow Architecture

Airflow supports a wide variety of deployment options. You can run Airflow on Kubernetes, your computer, in containers, on virtual machines or bare metal servers. This flexibility allows organizations to choose the deployment method that best fits their operational needs and existing infrastructure.

Running Airflow on Kubernetes is a popular choice due to the scalability and reliability that Kubernetes offers. Scaling Airflow to ensure it has the resources to run workflows at scale can be challenging. Using Kubernetes, you can easily scale Airflow components horizontally by increasing or decreasing the number of replicas for each component.

When deploying Airflow on Kubernetes, there are three key components:

- Scheduler
- Webserver
- Workers

In Kubernetes, each component runs as a pod. The scheduler and webserver support horizontal scaling. In a highly available Airflow cluster, you'd run multiple replicas of these components for scalability, high availability, and fault tolerance.

A crucial third component, not explicitly highlighted in the diagram, is Airflow's metadata database. This database is essential for persisting configuration details, task states, and other critical metadata. Airflow supports robust relational databases such as PostgreSQL and MySQL for this purpose. In production environments, it's considered a best practice to implement a highly available database cluster to ensure uninterrupted Airflow operations. We strongly recommend leveraging your cloud provider's managed database service for this purpose. For instance, if you're operating on AWS, Amazon RDS (Relational Database Service) offers an excellent solution. Once you have a database, you can reference it during installation. The list of all configuration parameter the Airflow Helm chart offers is available on GitHub: <https://github.com/apache/airflow/blob/main/chart/values.yaml#L448>.

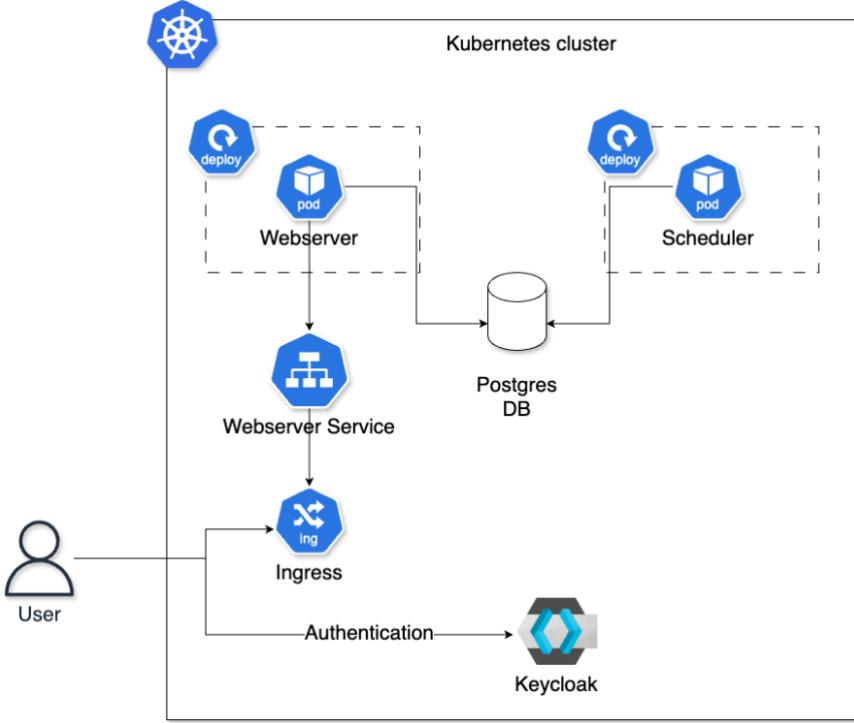


Figure 4.6 Airflow components on Kubernetes. Users accessing the Airflow web UI are authenticated using Keycloak

SCHEDULER

The *scheduler* is the component that parses, schedules, and monitors DAGs (which tells Airflow when and how to run a workflow). In Airflow, DAGs are written as Python files. A DAG contains one or more tasks, dependencies, and scheduling intervals. The scheduler extracts this information from DAG files. When the time comes to run a DAG, the scheduler tells the workers to execute them.

WORKERS

Workers are the processes that execute the tasks defined in the DAGs, once the scheduler has determined they should run. The workers pull tasks from the queue and run them as instructed by the executor.

Workers listen to one or more task queues and execute the tasks delivered by the scheduler. When a worker process starts, it specifies which queues it will listen to. This allows tasks to be assigned to specific queues, enabling workload

isolation and dedicated worker pools for different types of tasks. The executor determines how tasks are run by the workers.

The `KubernetesExecutor`, which is the preferred executor when running Airflow on Kubernetes, takes a different approach to worker scaling. Instead of maintaining a pool of pre-provisioned workers, it creates a dedicated Pod for each task. It eliminates the need to have long running workers. On Kubernetes, the Airflow cluster scales workers down to zero when there are no more tasks to run.

WEBSERVER

The webserver is the user interface component of Airflow that allows users to monitor, manage and troubleshoot DAGs and tasks. It is a Flask web application that renders the Airflow UI in a web browser.

The webserver provides different views to help you manage the Airflow cluster and monitor DAG executions. Airflow documentation includes a helpful guide to the various views that the webserver provides (<https://airflow.apache.org/docs/apache-airflow/stable/ui.html>).

4.4.4 Executors

Airflow executors bridge schedulers and workers, distributing tasks for execution. They come in two types: local (running tasks within the scheduler) and remote (running tasks on separate workers).

Remote executors, preferred in distributed environments like Kubernetes, offer better scalability and fault tolerance. The `KubernetesExecutor` is a remote executor that executes tasks in Pods, leveraging native scheduling, isolation, and resource management.

Advantages of `KubernetesExecutor` include:

- Simplified architecture compared to the traditional CeleryExecutor (no separate message broker)
- Efficient resource utilization and isolation
- Scalability and elasticity through auto-scaling
- Integration with Kubernetes features (Persistent Volumes, ConfigMaps, Secrets)

`KubernetesExecutor` works in conjunction with Kubernetes Cluster Autoscaling to save on costs. The cluster automatically scales up when Airflow schedules tasks (in Pods). When tasks finish, the Pods running them terminate, allowing the cluster to scale down.

Additionally, the `KubernetesExecutor` can leverage other Kubernetes features like Persistent Volumes, ConfigMaps, and Secrets, simplifying data and configuration management (we'll see this in action shortly).

4.4.5 Airflow Operators

A DAG in Airflow contains tasks that need execution, and Airflow operators (these aren't Kubernetes operators) simplify performing these tasks. Airflow simplifies hundreds of tasks using operators. How do you know which operator to pick? That depends on what you are trying to do. For example, let's say you have a task that downloads a file from an online source. You can do this in a couple of ways. You can write a bash script or a Python function to download the file.

Put simply, if you use a bash script, you'll go with the `BashOperator`. If it's a Python function, use the `PythonOperator`. Depending on your task, you'll pick an appropriate operator. Airflow allows mixing operators within a DAG, so you can download the file using a bash script and then run Python functions to process the file.

4.5 Creating your first DAG

Data pipeline development typically progresses from interactive Jupyter notebooks to structured Airflow DAGs for scheduled execution. A basic pipeline might include data ingestion and cleansing tasks, with the latter dependent on the former's completion.

Airflow's distributed architecture runs each step of a pipeline in a separate task. By doing so, Airflow improves its scalability and resource efficiency. By decoupling tasks and allowing for flexible scheduling, Airflow can optimize resource utilization and prevent bottlenecks, ultimately leading to more reliable and efficient data pipelines.

There are several ways to declare a DAG. We'll use context manager. You can read about other DAG declaration methods in Airflow documentation (<https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html>).

Listing 4.3 A DAG using a context manager

```
from airflow import DAG
from airflow.utils.dates import days_ago

with DAG(
    dag_id="my_dag", #A
    start_date=days_ago(1), #B
    schedule="@daily", #C
```

```

        catchup=False,
):
    task1 = ... #D
    task2 = ... #E

    task1 >> task2 #F
#A Define the name of your DAG
#B The first date when Airflow should run this DAG
#C The task should run everyday
#D The definition of the first task
#E The definition of the second task
#F The order in which the tasks should be executed

```

Consider the sample DAG shown in listing 4.3. This DAG contains two tasks. We've expressed that "task1" should run first, followed by "task2". Within a DAG, the order in which Airflow should execute tasks is defined using ">>" or the right shift operator.

Let's say we want "task1" to print the current date (using the `date` command). This is a simple enough task, and we can do this using bash. We can run `task1` using `BashOperator`, defining it using the following syntax:

```

from airflow.operators.bash import BashOperator

task1 = BashOperator(
    task_id="print_current_date",
    bash_command="date"
)

```

When Airflow runs this task using `KubernetesExecutor`, the Airflow scheduler creates a Pod in which the `date` bash command is run.

While shell commands are good for simple tasks, more complicated tasks will benefit from having the flexibility that Python offers. To run a Python function, you use the `PythonOperator`. First, you write the function logic and then reference it in the task.

```

from airflow.operators.python import PythonOperator

def my_python_function():
    print("Hello from my_python_function!")

task2 = PythonOperator(
    task_id="python_task",
    python_callable=my_python_function,
)

```

The DAG with the two steps is shown in listing 4.4.

Listing 4.4 A complete DAG

```

from airflow import DAG
from airflow.utils.dates import days_ago
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator

def my_python_function():
    print("Hello from my_python_function!")

with DAG(
    dag_id="my_dag",
    start_date=days_ago(1),
    schedule="@daily",
):
    task1 = BashOperator(
        task_id="print_current_date",
        bash_command="date"
    )
    task2 = PythonOperator(
        task_id="python_task",
        python_callable=my_python_function,
    )

    task1 >> task2

```

The `start_date` parameter in Airflow determines when the DAG will start scheduling. If the `start_date` is set to a past date, Airflow will schedule DAG runs for all unprocessed data intervals since that date.

In Airflow, this behavior is known as "catchup" or backfilling and is controlled by the `catchup` parameter in the DAG definition. By default, `catchup` is set to `True`. We don't want Airflow to kick off DAG runs for missed intervals, so we should set `catchup=False` in our DAG definition:

```

...
with DAG(
    dag_id="my_dag",
    start_date=datetime(2024, 1, 1),
    schedule="@daily",
    catchup=False,
):
...

```

You can also define an end date for your DAG, beyond which Airflow won't schedule DAG runs.

Airflow DAGs are flexible to serve virtually every data engineering need. Airflow also offers fine grained control over how DAGs and tasks within them run. For example, you can control task concurrency, configure callbacks, specify retry behaviors, and more. These topics are beyond the scope of this book. We recommend Bas P. Harenslak and Julian Rutger de Ruiter's *Data Pipelines with*

Apache Airflow (Manning 2021). For the rest of this chapter, we'll focus on how you can run a scalable Airflow deployment on Kubernetes.

4.6 Loading DAGs in Airflow

To run a DAG in Airflow, you must make the DAG available to Airflow. There are three ways to do this:

- Container images – You can create a custom Airflow container image(s) and bake your DAGs into the image.
- Shared volume – You use a network file share to store DAGs.
- Git-Sync – You can store DAGs in a Git repository and have Airflow sync the DAGs from Git at regular intervals.

Using container images is the least preferred approach as it requires rebuilding images every time there's a change to the DAG. Git-Sync is a viable approach when DAGs are stored in Git repositories. Git-Sync can be very useful when you have a CI/CD pipeline that lints and tests DAG files before making them available to Airflow. As your team's usage grows, Git-Sync with CI/CD should become the preferred way.

The easiest way to upload DAGs to Airflow is using a shared directory approach. With shared directory, users can copy DAGs to a shared folder and run them using Airflow. We'll use this approach. In chapter 3, we created a shared directory using Amazon EFS. It is mounted on notebook servers at `/home/shared` using a persistent volume. To make DAGs available to Airflow, copy DAG files to the file share.

NOTE You can mount persistent volumes provided by Amazon EFS across multiple Pods. This differs from Amazon EBS, where you can only mount a volume to one Pod at a time.

In the `airflow-values.yaml` file, you'll notice two mount points that map specific subdirectories from the Amazon EFS file system to their respective paths within the Airflow container. To Airflow, these directories seem just like any other local filesystem directories.

The first mount point is:

```
volumeMounts:  
  - name: airflow-shared  
    mountPath: /opt/airflow/dags  
    subPath: airflow_home/dags
```

This mount point maps the `/airflow_home/dags` subdirectory from the EFS file system to the `/opt/airflow/dags` path within the Airflow container. This is the default location where Airflow looks for DAG files.

The second mount point is:

```
- name: airflow-shared
  mountPath: /var/log/airflow
  subPath: airflow_home/logs
```

This mount point maps the `/airflow_home/logs` subdirectory from the Amazon EFS file system to the `/var/log/airflow` path within the Airflow container. This is the default location where Airflow stores log files.

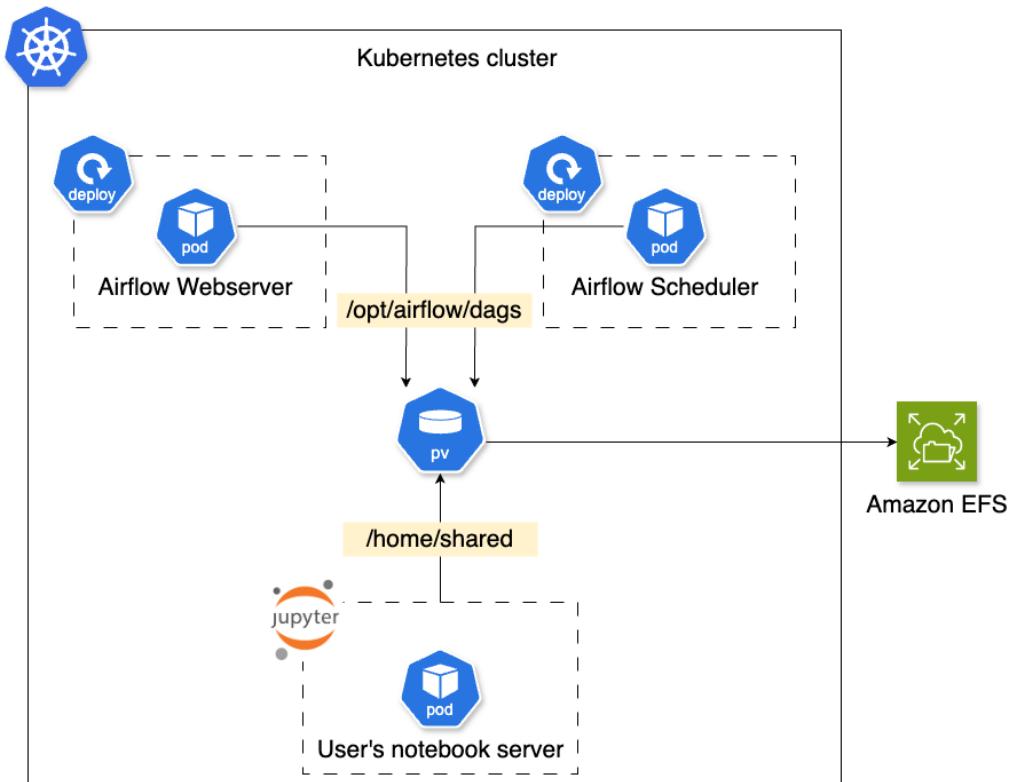


Figure 4.7 All Airflow components have access to a network file share. Notebook users can use this file share to send DAGs to Airflow.

After installing Airflow, the Amazon EFS share file system has a directory called `airflow_home`, which contains two subdirectories:

```
/  
└── airflow_home  
    ├── dags  
    └── logs
```

The `airflow_home/dags` subdirectory is where you can place your DAG files, and the `logs` subdirectory is where Airflow will store log files for your data pipelines.

By configuring these mount points, we've effectively decoupled the DAG files and log files from the Airflow container itself. This shared filesystem is available to all Airflow Pods, even if they run on separate worker nodes.

4.7 Running your first DAG

Using a Jupyter notebook, you can now create a file for the DAG shown in listing 4.4 and copy the file to `/home/shared/airflow_home/dags/`. Airflow scans the `dags` directory periodically for new DAGs.

Once Airflow detects the DAG, it will appear in the Airflow dashboard. By default, Airflow scans for new DAGs every 30 seconds.

The screenshot shows the Airflow web interface with the following details:

- Header:** Airflow, DAGs, Cluster Activity, Datasets, Security, Browse, Admin, Docs, 20:24 UTC, User icon.
- Section:** DAGs
- Filter Buttons:** All (1), Active (0), Paused (1), Running (0), Failed (0).
- Search:** Filter DAGs by tag, Search DAGs.
- Actions:** Auto-refresh, Refresh icon.
- Table Headers:** DAG, Owner, Runs, Schedule, Last Run, Next Run, Recent Tasks, Actions, Links.
- Table Data:** One row for "my_dag" owned by "airflow". The "Schedule" column shows "@daily". The "Last Run" column shows "2024-06-19, 00:00:00". The "Recent Tasks" column shows a series of small circles representing recent runs. The "Actions" column has a play icon.
- Pagination:** Showing 1-1 of 1 DAGs, with a page number indicator at 1.

Figure 4.8 Airflow web user interface

By default, all new DAGs are paused. Airflow will not schedule DAG runs for paused DAGs. You can unpause a DAG using the 'On/Off' toggle button located besides name of the DAG.

To trigger a DAG, click on the play icon in the "Actions" column. Triggering a paused DAG automatically unpauses it. Once you trigger this DAG, Airflow will

start executing the two tasks in Pods. In fact, Airflow will execute each task in a different Pod.

Once the DAG run completes, you can view logs in the Airflow UI.

The screenshot shows the Airflow web interface for a DAG named "my_dag". A specific task, "print_current_date", is selected. A callout points to this task with the text "Select a task". Another callout points to the "Logs" tab in the top navigation bar with the text "Switch to the Logs tab". The "Logs" tab is currently active, displaying log entries for the task. The log output includes:

```

my-dag-print-current-date-sth4k12c
*** Found local files:
***   = /var/log/airflow/dag_id=my_dag/run_id=manual__2024-06-19T20:37:01.364877+00:00/task_id=print_current_date/attempt=1.log
[2024-06-19, 20:37:00 UTC] (taskinstance.py:1979) INFO - Dependencies all met for dep_context<non-requeuable deps t=TaskInstance: my_dag.print_current_
[2024-06-19, 20:37:00 UTC] (taskinstance.py:1979) INFO - Dependencies all met for dep_context<non-requeuable deps t=TaskInstance: my_dag.print_current_
[2024-06-19, 20:37:00 UTC] (taskinstance.py:2217) INFO - Executing <Task(BashOperator): print_current_date> on 2024-06-19 20:37:01.364877+00:00
[2024-06-19, 20:37:00 UTC] (standard_task_runner.py:60) INFO - Started process 19 to run task
[2024-06-19, 20:37:00 UTC] (standard_task_runner.py:87) INFO - Running: ['airflow', 'tasks', 'run', "my_dag", 'print_current_date', 'manual__2024-06-19T20:37:01.364877+00:00']
[2024-06-19, 20:37:00 UTC] (standard_task_runner.py:87) INFO - Job 122: Subtask print_current_date
[2024-06-19, 20:37:00 UTC] (task_command.py:42) INFO - Running: <TaskCommand: my_dag.print_current_
[2024-06-19, 20:37:00 UTC] (logging_mixin.py:188) WARNING - /home/airflow/.local/lib/python3.8/site-packages/airflow/providers/cncf/kubernetes/template_
[2024-06-19, 20:37:00 UTC] (pod_generator.py:555) WARNING - Model file /opt/airflow/pod_templates/pod.yaml does not exist
[2024-06-19, 20:37:00 UTC] (taskinstance.py:2513) INFO - Exporting env vars: AIRFLOW_CTX_DAG_OWNER='airflow' AIRFLOW_CTX_DAG_ID='my_dag' AIRFLOW_CTX_T_
[2024-06-19, 20:37:00 UTC] (subprocess.py:63) INFO - Tmp dir root location: /tmp
[2024-06-19, 20:37:00 UTC] (subprocess.py:63) INFO - Command: ['!/usr/bin/bash', '-c', 'date']
[2024-06-19, 20:37:00 UTC] (subprocess.py:63) INFO - Output:
[2024-06-19, 20:37:00 UTC] (subprocess.py:93) INFO - Wed Jun 19 20:37:00 UTC 2024
[2024-06-19, 20:37:00 UTC] (subprocess.py:97) INFO - Command exited with return code 0
[2024-06-19, 20:37:00 UTC] (taskinstance.py:1149) INFO - Marking task as SUCCESS, dag_id=my_dag, task_id=print_current_date, execution_date=20240619T20:37:01.364877+00:00
[2024-06-19, 20:37:00 UTC] (local_task_job_runner.py:234) INFO - Task exited with return code 0
[2024-06-19, 20:37:00 UTC] (taskinstance.py:3312) INFO - 1 downstream tasks scheduled from follow-on schedule check

```

Figure 4.9 Airflow captures task logs and displays them in the Airflow dashboard. To view logs, select a task and switch to the Logs tab.

4.8 Passing data between tasks

So far, the tasks in our DAG operate independently, without exchanging information. Sometimes you'll need to pass information from a task to downstream tasks. Airflow provides several mechanisms to facilitate data sharing between tasks within a DAG.

One common approach is to use XComs (short for "cross-communication"). XComs allow you to push and pull small amounts of data (e.g., metadata, scalar values, or small data structures) between tasks. Within a task, you can push a key-value pair using the `xcom_push` method. You can then read the value from another task using the `xcom_pull` method. The data pushed to XComs should

be JSON-serializable. Complex objects or large data structures might need to be serialized before pushing and deserialized after pulling. Listing 4.5 shows an example DAG in which task1 pushes data into XCom and task2 reads it.

First, we explicitly configure `do_xcom_push=True` in the `BashOperator`. This is purely for illustration as the value is set to True by default. With this setting, the bash command's last line is automatically pushed to an XCom once the command completes. In fact, you can see the return values from previous DAGs you've run by going to Admin > XComs in the Airflow web UI.

Second, we accept the `context` argument in `task2`. The `context` in Airflow is a dictionary containing crucial information about a task's execution environment. It includes details about the current task instance, the DAG, the execution environment, and XComs. This information can be useful for various purposes, such as logging, conditional execution, and passing data between tasks. Many Airflow operators automatically push a task's return value to XCom.

When a task pushes an XCom, Airflow stores the task's XCom key-value pair in its metadata database. However, this method has inherent limitations when dealing with large data values. Databases often have size constraints for individual records, which directly impacts the maximum size of XComs. XComs are not suitable for storing or transferring large values. For instance, using XComs to send a large CSV file from one task to another would not be a good practice.

Listing 4.5 DAG with Xcom

```
from airflow import DAG
from airflow.utils.dates import days_ago
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator

def my_python_function(**context): #A
    xcom_value =
context['ti'].xcom_pull(task_ids='print_current_date') #B
    print(f"value from task1 = {xcom_value}")

with DAG(
    dag_id="dag_with_xcom",
    start_date=days_ago(1),
    schedule="@daily",
    catchup=False,
):
    task1 = BashOperator(
        task_id="print_current_date",
        bash_command="date",
        do_xcom_push=True #C
    )
    task2 = PythonOperator(
```

```

        task_id="python_task",
        python_callable=my_python_function,
        provide_context=True #D
    )

    task1 >> task2
#A The Python function receives context as an argument
#B The function reads XCom value from the context
#C The BashOperator is set to push command output to XCom
#D The PythonOperator includes context when calling the Python function

```

When using Postgres, an XCom can be as large as 1 GB. MySQL limits XComs to 64KB. These constraints highlight the importance of considering alternative methods for large data transfers between tasks, such as shared file systems or external data stores.

If your tasks need to pass larger amounts of data (like data set files), you have two options.

First, you can use a custom XCom backend. Airflow supports object storage backend (like Amazon S3). By leveraging custom XCom backends, you can overcome the size limitations of traditional database-stored XComs while maintaining the simplicity of the XCom API.

Option two for inter-task data transfer: Use an intermediate system like a shared filesystem, cloud storage, message broker (like Kafka), or database (like Postgres or Redis) to read and write data within the pipeline code, thus bypassing XCom entirely.

The latter option is easier to implement since we already have a shared filesystem available in our Kubernetes cluster.

4.8.1 Passing large amounts of data between tasks

When converting a data processing notebook into an Airflow pipeline, a key challenge emerges: the inability to directly pass large DataFrames between tasks.

In a Jupyter notebook, it's straightforward to perform a series of operations on a DataFrame, with each step building upon the results of the previous one. However, this seamless flow doesn't translate directly to Airflow's task-based structure.

Airflow's design, optimized for orchestration rather than data processing, doesn't support passing large in-memory objects like DataFrames between tasks. This constraint requires that data engineers:

1. Break down the process into discrete, independent tasks.
2. Use intermediate storage (like files or databases) to pass data between tasks.

3. Redesign workflows to load, process, and save data at each step.

This shift in approach requires careful planning but ultimately leads to more robust, scalable, and maintainable data pipelines. It also aligns well with best practices in data engineering, promoting modularity and allowing for easier monitoring and error handling at each stage of the process.

Let's explore this by using a shared persistent volume for task data storage. Going with a custom XCom backend approach would follow a similar pattern, with the fundamental difference being that tasks would use XCom to read and write data instead of interacting with the local filesystem.

Recall that, in our setup, Airflow Pods have a shared persistent volume mounted at `/var/log/airflow` (for log storage) and `/opt/airflow/dag` (for DAG storage). This volume is accessible from all Airflow Pods, including the Pods that run DAGs. While we can use one of these paths to store dataset files we process in DAGs, a better option will be to create a dedicated location for data downloaded and passed through a pipeline.

Let's say we want the pipeline steps to write outputs to the `/pipeline-data` path of the local filesystem. Since each task runs in a Pod, we can create another mountpoint to mount shared persistent volume at `/pipeline-data`.

The Airflow Kubernetes executor allows us to configure the Pod in which tasks run. We can use this feature to mount the shared filesystem at `/pipeline-data` as well. Listing 4.6 shows how you can use the `pod_override` option to add another mountpoint.

Listing 4.6 Adding a mount point by overriding Pod defaults

```
with DAG(...) :
    executor_config = {
        "pod_override": k8s.V1Pod(
            spec=k8s.V1PodSpec(
                containers=[
                    k8s.V1Container(
                        name="base",
                        volume_mounts=[

                            k8s.V1VolumeMount(mount_path="/pipeline-data/", #A
                                              name="airflow-shared", #B
                                              sub_path="pipline-data") #C
                        ],
                    )
                ]
            )
        )
    }
```

```

task1 = BashOperator(
    executor_config=executor_config,
    ...
)
#A Mount path
#B Name of the persistent volume
#C Sub path of the persistent volume

```

Airflow uses Python Kubernetes client libraries to override Pod defaults. This allows you to control a Pod configuration from within a DAG's code.

4.9 Transitioning from notebook to pipeline

Let's use the medical drug dataset example we started this chapter with to understand the process of transitioning code from a Jupyter notebook to a data pipeline.

We wrote the code to:

20. Download the dataset from Kaggle
1. Remove unneeded columns
2. Fix missing values

Our DAG will thus have three tasks.

Listing 4.7 DAG with three tasks

```

import ...

with DAG(
    ...
) :
    download_and_extract_data=...
    remove_columns=...
    remove_missing_values=...

    download_and_extract_data >> remove_columns >>
    remove_missing_values

```

The first task in the pipeline downloads data from Kaggle. This task presents two challenges:

1. The Airflow image doesn't have `kaggle` installed.
2. How shall we securely provide the task our Kaggle authentication information?

We obviously won't hard code our precious API keys.

4.9.1 Providing secrets to tasks

Recall that Kaggle APIs require authentication before we can download the dataset. Therefore, we must provide the API key details to the task.

There are several ways to provide data to a task:

- As direct parameters to the task
- Stored in Airflow's metadata store and retrieved by tasks
- Stored as Kubernetes secrets and injected into tasks

Providing data as parameters is the simplest approach to implement, as shown in listing 4.8. Airflow supports DAG-level and task-level parameters. However, with this approach you're storing sensitive data in your DAG. Anyone that has access to your DAG file can view your Kaggle username and key.

Listing 4.8 Providing parameters to tasks

```
def print_my_int_param(params):
    print(params.my_int_param)

PythonOperator(
    task_id="print_my_int_param",
    params={"my_int_param": 10},
    python_callable=print_my_int_param,
)
```

Since Kubernetes already provides us a secure way of storing secrets, we'll store our Kaggle username and password as Kubernetes secrets instead of hard coding them into our DAG code.

Kubectl makes it easier to create secrets from a file. You can create a secret using the `kaggle.json` file (which contains your username and key):

```
$ kubectl create secret generic kaggle \
--from-file=/path/to/kaggle.json \
-n airflow
```

We can now mount this secret as a volume.

Listing 4.9 Mounting a secret as file in a task Pod

```
from airflow.kubernetes.secret import Secret

kaggle_secret = Secret(
    deploy_type="volume", #A
    deploy_target="/home/airflow/.kaggle/", #B
    secret="kaggle", #C
    key="kaggle.json", #D
)
```

```
#A Mount the secret as a volume (the other option is to create environment variables)
#B The location where kaggle.json should be stored
#C The name of the secret
#D The key that contains kaggle.json contents within the secret
```

NOTE Kubernetes also integrates with external secret management systems like HashiCorp Vault, AWS Secrets Manager, Azure Key Vault, and Google Secrets Manager. This integration is provided by External Secrets Operator (<https://external-secrets.io/>). The operator reads information from external APIs and injects the value into Kubernetes secrets.

To inject this secret into a Pod, we can once again use the `pod_override` parameter shown in listing 4.6. Now that we have a way to provide secret data to tasks, let's find out how we can run the Kaggle CLI in an Airflow task.

4.9.2 Handling dependencies

When using the `KubernetesExecutor`, Airflow runs tasks inside Pods, which in turn run inside containers using the Airflow container image. Sometimes tasks in your data pipeline require dependencies that are unavailable in the default Airflow container image.

For example, the data ingestion step in our pipeline requires the `kaggle` binary. In the notebook, we simply executed `pip install kaggle` to install it. Since this dependency is not installed in the default Airflow container image, we must create a new Airflow image that includes any task dependencies. Listing 4.10 shows a Dockerfile that is used to create an Airflow image with the Kaggle API CLI installed.

Listing 4.10 Dockerfile for building a custom Airflow image

```
FROM apache/airflow:2.8.3

USER root

RUN apt-get update \
    && apt-get install -y --no-install-recommends \
    unzip \
    && apt-get autoremove -yqq --purge \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

USER airflow

RUN pip install kaggle
```

Besides installing packages using pip, a custom image can also include packages installed from other sources (like a Linux package repository).

Build your custom image using the Docker build command:

```
$ docker build -t my-custom-airflow:latest .
```

NOTE You don't have to create this custom image. The authors have created an image and made it available on DockerHub (<https://hub.docker.com/repository/docker/realz/airflow-custom>)

We can then tell Airflow to run tasks using this custom image with the help of `KubernetesPodOperator`, which is an Airflow operator that lets you use custom container images. It is useful when tasks have dependencies that are unavailable in the standard Airflow container image. You can even use it to execute tasks in languages other than Python.

Using `KubernetesPodOperator`, we create a task to download and extract the dataset.

Listing 4.11 A task using KubernetesPodOperator

```
download_and_unzip = KubernetesPodOperator(  
    ...  
    image="realz/airflow-custom:v1",  
    secrets=[kaggle_secret],  
    ...  
)
```

`KubernetesPodOperator` allows you to override Pod defaults and add resources to the task. Listing 4.12 contains a snippet of a DAG in which we use `KubernetesPodOperator` to mount the shared volume at `/pipeline-data` on the task's local filesystem.

Listing 4.12 A DAG to download and extract dataset (Chapter 4/airflow/DAG/medi-drug-prep.py)

```
...  
    kaggle_secret = Secret( #A  
        deploy_type="volume",  
        deploy_target="/home/airflow/.kaggle/",  
        secret="kaggle",  
        key="kaggle.json",  
    )  
  
    volume = k8s.V1Volume( #B  
        name="airflow-shared",  
  
        persistent_volume_claim=k8s.V1PersistentVolumeClaimVolumeSource(  
            claim_name="airflow-efs-shared"),
```

```

        )

volume_mount = k8s.V1VolumeMount( #C
    name="airflow-shared", mount_path="/pipeline-data/",
    sub_path="pipeline-data"
)

download_and_unzip_args=( "kaggle datasets download -d
himalayaashish/medi-drug-dataset -p ~/;""
                           "unzip -o ~/medi-drug-dataset.zip -d
/pipeline-data/;""
                           )

with DAG(
    dag_id="med-drug-prep",
    start_date=days_ago(1),
    schedule="@daily",
    catchup=False,
) :

    download_and_unzip = KubernetesPodOperator(
        task_id="download_and_unzip",
        name="download_and_unzip",
        image="realz/airflow-custom:v1",
        cmd=["sh", "-c"],
        arguments=[download_and_unzip_args],
        secrets=[kaggle_secret], #D
        volumes=[volume], #E
        volume_mounts=[volume_mount], #F
        get_logs=True,
        is_delete_operator_pod=True,
    )
#A Define the secret that contains Kaggle username and key
#B Declare a Persistent Volume
#C Declare a mount point for the Persistent Volume
#D Add the secret to the Pod
#E Add Persistent Volume
#F Add Persistent Volume mount point

```

When this DAG runs, it downloads the raw dataset and stores it on the shared drive. It is a best practice to write the intermediate results of data pipelines on to persistent storage. Saving processed data to storage between tasks creates checkpoints in the pipeline. This is a key strategy for building robust data workflows. In case of failures, checkpoints allow you to resume processing from the last successful step, rather than starting from scratch. This saves time and computational resources. Remember to create a process that purges orphaned checkpoint files from failed jobs (perhaps a Kubernetes CronJob or another Airflow DAG?).

4.9.3 Adding data processing step to the pipeline

With the pipeline set up to download and extract the dataset, we can add a data processing task. Using Python for processing, we simply need a Python function and a `PythonOperator`-based task to execute it. The data processing steps we performed in the notebook can be expressed as:

```
def _process_data():
    df = pd.read_csv('/pipeline-data/Medi_Drug.csv')
    df.drop(columns='Information', inplace=True)
    df['Indication'] = df['Indication'].str.replace('\r\n',
    'Unidentified')
    df.to_csv('/pipeline-data/pipeline-output.csv', index=False)
```

To run this Python function, we use the `PythonOperator`:

```
process_data = PythonOperator(
    task_id="process_data",
    python_callable=_process_data,
    executor_config=executor_config
)
```

With this, we now have the pipeline to ingest and process the data. The DAG is available on the book's GitHub repo at `Chapter 4/airflow/DAG/medi-drug-prep.py`.

When this DAG runs, the processed output is stored at `/pipeline-data/pipeline-output.csv` (or at `/home/shared/pipeline-data/pipeline-output.csv` if you're accessing it from a Jupyter notebook).

To recap what the pipeline does:

- It downloads the dataset from Kaggle.com using `kaggle CLI`. In the same task, the data is unzipped and stored at `/home/shared/pipeline-data/Medi_Drug.csv`.
- The second task uses a Python function to read the CSV file located at `/home/shared/pipeline-data/Medi_Drug.csv`, processes it, and stores the output at `/home/shared/notebook-output.csv`.

This experiment should give you a fair idea on how on how you go from a notebook to a pipeline. Here is a quick recap:

1. Modularize notebook code into discrete, reusable functions.
2. Convert notebook variables into task parameters.
3. Pick the best operator for the task, preferably that requires the least amount code.
4. Determine how data flow occurs between tasks.
5. Add additional resources (like a Secret or GPU).
6. Setup a schedule to run the pipeline.

4.10 Configuring the default properties of worker Pods

In our previous section, we added a Persistent Volume to all worker Pods using the `pod_override` option. This option is suitable for task-specific configurations. You can also set global defaults for worker Pods using a *Pod template file*.

For example, if you want all Airflow task Pods to mount a Persistent Volume, instead of adding the volume using `pod_override` in each DAG, you can create a *Pod template file*, which the Kubernetes executor will use as the template for all worker Pods.

Pod templates are useful when you want every Pod that the Kubernetes executor creates to have specific properties. This can be useful when every Pod needs a persistent volume or must have a label.

There are two properties that you must include in Pod template files:

- The first container in the template file must be named `base` and it must have an image specified.
- The template must include a Pod name (`metadata.name`).

Listing 4.14 A minimal Pod template file

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-name #A
spec:
  containers:
    - name: base #B
      image: <image>:<tag> #C
      volumeMounts:
        ...
  volumes:
    ...
#A KubernetesExecutor will dynamically generate a unique Pod name
#B The first container must be named base
#C The base container must be specified
```

The Airflow Helm chart can generate a Pod template file during installation. The Airflow GitHub repository contains a Pod template file (<https://github.com/apache/airflow/blob/main/chart/files/pod-template-file.kubernetes-helm-yaml>) that you can customize to suit your needs.

4.11 Managing resources

Airflow on Kubernetes utilizes infrastructure resources more efficiently than the traditional setup with servers. In the past, DevOps teams provisioned a fixed number of long-running workers. These workers consumed resources even when there weren't any tasks to run. On Kubernetes, Airflow doesn't need long running workers. Thanks to the Kubernetes executor, Airflow creates workers only a DAG submits a task.

By default, tasks are run in Pods that have no dedicated CPU, memory, or GPU resources. Nor do they have any resource consumption limits. In chapter 2, we discussed how you can allocate and restrict resources at Pod and container level using requests and limits.

Running workloads without declaring resource constraints is a recipe for disaster. Without resource constraints, application performance in a Kubernetes cluster is best-effort, meaning there's no guarantee of reliable operation, especially during periods of resource pressure. Requests help the Kubernetes scheduler make informed decisions about Pod placement, ensuring nodes have sufficient resources to run Pods. Limits prevent a single Pod from consuming all available resources on a node, which could starve other Pods.

Configuring requests and limits ensures that your cluster's performance is more predictable. By setting appropriate requests, you ensure that pods have the minimum resources they need to function properly. Limits help maintain consistent performance by preventing resource contention between pods. Limits are used to prevent the noisy neighbor problem, which occurs when a misbehaving application impacts the performance of other applications running on the same node.

You have two options for configuring resource requirements for tasks:

- At task level using custom `executor_config`
- At global level by using a Pod template file

Get the best of both worlds by setting default resources using a Pod template file and override it for tasks that need more resources using `executor_config`.

Listing 4.15 shows a PythonOperator with requests and limits.

Listing 4.15 Specifying per task resource requests and limits

```
task = PythonOperator(  
    ...  
    executor_config={  
        "pod_override":  
            k8s.V1Pod(spec=k8s.V1PodSpec(containers=[  
                k8s.V1Container(  
                    name="base",  
                    resources=k8s.V1ResourceRequirements(  
                        limits={  
                            "cpu": "1",  
                            "memory": "1Gi"  
                        },  
                        requests={  
                            "cpu": "0.5",  
                            "memory": "512Mi"  
                        }  
                    )  
                ]  
            )  
        }  
    }  
)
```

```
        requests={"cpu": "100m", "memory": "256Mi"},  
        limits={"memory": "420Mi"}  
    )  
)  
])  
}  
}
```

Besides specifying task level resources, consider limiting the total number of resources Airflow can create. The goal is to ensure that a misconfigured DAG or task doesn't consume excessive resources. There are two options to limit resources Airflow tasks can request:

- Airflow Pools
 - Kubernetes Resource Quotas

Once again, you can combine these to have multiple safeguards.

4.11.1 Airflow Pools

Picture a scenario where your data platform serves multiple teams: marketing and finance. Each team has its own set of data pipelines, all clamoring for resources in a cluster that's scalable yet without boundless resources. In such a multitenant environment, having fair resource sharing is a prerequisite to ensure smooth operation.

In Airflow, pools allow data engineers and DevOps teams to bring order to the chaos of concurrent task execution. At its core, a pool is a named entity with a fixed number of slots, each representing a unit of computational resource. A pool limits execution parallelism for a group of tasks.

Without pools, you might find the finance team's critical end-of-month reports stuck behind a queue of less urgent marketing email campaigns. Pools allow you to allocate resources fairly, ensuring that each team gets its fair share of the computational swim lane.

Creating a pool is as simple as giving it a name and assigning it a number of slots. For instance, you might create a "finance_pool" with 10 slots and a "marketing_pool" with 5 slots. Tasks are then assigned to these pools, and Airflow's scheduler becomes the traffic controller, managing the flow of task execution across your entire platform. After creating a pool, a task can be assigned to a pool in this manner:

```
task = BashOperator(  
    task_id="finance_task",  
    ...  
    pool="finance"  
)
```

Each task uses one slot by default. Tasks that do relatively heavier tasks can be allocated multiple slots to ensure the underlying system doesn't starve for resources. But you're on Kubernetes, so you also can easily adjust a task's resources.

4.11.2 Limiting resources in Kubernetes

Kubernetes is designed with multitenancy in mind, more specifically, enabling multiple teams to operate their workloads in a shared cluster environment without resource contention. This approach offers many benefits, including improved resource utilization, simplified management, and cost-effectiveness. However, with shared resources comes the challenge of ensuring fair allocation and preventing any single team or application from monopolizing the cluster's capacity. This is where Kubernetes Resource Quotas come into play, providing a constraint to limit the aggregate resource consumption per namespace.

Think of namespaces as virtual sub-clusters within your Kubernetes environment, each potentially representing a different team, project, or application. By applying Resource Quotas to these namespaces, you can set hard limits on various resources, ensuring that each tenant operates within their allocated boundaries.

Resource Quotas apply to a variety of resources. They can limit not only compute resources like CPU and memory but also storage resources and even the number of Kubernetes objects that can be created. For instance, you can restrict the total number of Pods, Services, Config Maps, or Persistent Volume Claims within a namespace. This granular control helps prevent scenarios where one team's overzealous use of resources impacts the performance and stability of other workloads in the cluster.

Listing 4.16 Resource Quota to limit CPU, memory, and GPU resources

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-quota
  namespace: example-namespace
spec:
  hard:
    requests.cpu: "100"
    limits.cpu: "200"
    requests.memory: 10Gi
    limits.memory: 20Gi
    requests.nvidia.com/gpu: 4
    pods: "100"
```

Resource Quotas work hand in hand with other Kubernetes concepts. For example, they interact closely with Limit Ranges, which set default resource limits for containers within a namespace. This combination ensures that not only are overall namespace resources capped, but individual containers within that namespace also adhere to sensible defaults.

Listing 4.17 Setting default requests and limits for Pods in a namespace

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-resource-constraint
spec:
  limits:
    - default:
        cpu: 500m
      defaultRequest:
        cpu: 500m
    max:
      cpu: "1"
    min:
      cpu: 100m
  type: Container
```

While it's not uncommon for developers to deploy workloads without specifying resource constraints, this practice can lead to resource contention and unpredictable performance in a Kubernetes cluster. In such situations, Limit Ranges help you configure the default requests and limits for workloads and set upper and lower bounds for resource allocation. By implementing Limit Ranges, you can:

- Ensure baseline performance for all workloads
- Prevent resource hogging by individual containers
- Improve cluster stability and resource utilization
- Encourage better resource planning among development teams

4.12 Using Apache Spark in pipelines

When dealing with massive amounts of data, data engineers often turn to Apache Spark as a tool for processing data. There are several compelling reasons Spark is one of the most popular tools for data analytics.

First, Spark's distributed computing model scales across a cluster of machines, handling massive datasets beyond single-server capacity. Python

libraries like Pandas and NumPy operate on a single machine, which means they are limited by the available memory and processing power of that machine.

Moreover, Spark's unified engine provides a comprehensive ecosystem for data processing, supporting batch processing, real-time streaming, machine learning, and graph computations all within a single framework. This versatility allows data engineers to build complex, end-to-end data pipelines without having to switch between multiple tools or platforms. Spark's ability to handle both batch and streaming data makes it particularly well-suited for building robust ETL (Extract, Transform, Load) and ELT (Extract, Load, Transform) pipelines that can adapt to changing data volumes and velocities.

It's in-memory processing capabilities significantly speed up iterative algorithms and interactive data analysis, making it much faster than traditional MapReduce-based systems. Spark's integration with various data sources and sinks, including HDFS, Hive, Kafka, and cloud storage services, makes it a versatile choice for building data pipelines that need to interact with diverse data sources.

Many organizations run Spark on Kubernetes. One of the key advantages of running Spark on Kubernetes is the ability to dynamically scale resources based on workload demands. Kubernetes can automatically scale Spark application, allowing them to efficiently handle varying data volumes and processing requirements. This elasticity is particularly valuable in cloud environments, where it can lead to significant cost savings by avoiding over-provisioning of resources.

4.12.1 *Running Spark jobs on Kubernetes*

Spark's support for Kubernetes has gone through several iterations. As a result, there are several ways to orchestrate Spark applications in a Kubernetes environment. We'll focus on Spark Operator, which simplifies the deployment and management of Spark applications on Kubernetes by extending the Kubernetes API with custom resources for Spark.

In Kubernetes, an *operator* combines custom resources and controllers to maintain the state of workloads it manages. The Spark Operator manages the state of Spark Applications in a Kubernetes cluster. It extends the Kubernetes API with custom resources for Spark, namely `SparkApplication` and `ScheduledSparkApplication`. These custom resources allow users to define Spark jobs declaratively, using YAML files, much like other Kubernetes resources. This declarative approach aligns with Kubernetes' philosophy of infrastructure as code, enabling users to manage Spark applications with the same tools and workflows they use for other Kubernetes workloads. It wraps the `spark-submit` binary to launch Spark applications.

The Spark Operator can be deployed using Helm:

```
$ helm repo add spark-operator https://kubeflow.github.io/spark-operator
$ helm repo update
$ helm install so spark-operator/spark-operator \
    --namespace spark-operator \
    --create-namespace \
    --set webhook.enable=true
```

Listing 4.18 shows a sample `SparkApplication`. This is not a distributed application. Running a distributed Spark application requires Hadoop storage. In cloud environments, object storage services, like Amazon S3, functions as the distributed storage layer. To eliminate this dependency, we've chosen this simpler example instead of a distributed one.

Listing 4.18 A sample `SparkApplication` manifest

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: pyspark-pi #A
  namespace: spark-operator #B
spec:
  type: Python #C
  pythonVersion: "3"
  mode: cluster
  image: "spark:3.5.0"
  mainApplicationFile:
    local:///opt/spark/examples/src/main/python/pi.py #D
    sparkVersion: "3.5.0"
  restartPolicy: #E
    type: OnFailure
    onFailureRetries: 3
    onFailureRetryInterval: 10
    onSubmissionFailureRetries: 5
    onSubmissionFailureRetryInterval: 20
  driver:
    cores: 1
    coreLimit: "1200m"
    memory: "512m"
    labels:
      version: 3.5.0
      serviceAccount: so-spark
  executor:
    cores: 1
    instances: 1 #F
    memory: "512m"
    labels:
      version: 3.5.0
#A Name of the SparkApplication resource
#B Namespace in which to create the resource
#C Type can be Python, Scala, Java, or R
#D The file that contains application code
```

```
#E Retry policies  
#F The number of executors to run
```

You create a `SparkApplication` using `kubectl`:

```
$ kubectl apply -f spark-py-pi.yaml
```

When you create a `SparkApplication`, the operator automatically runs `spark-submit` on your behalf. It creates two types of Pods:

- Driver – A central process that coordinates and manages Spark applications
- Executors – Workers that perform data processing tasks

A Spark application here refers to a unit of work, such as a Spark jar, PySpark script, or SparkSQL query. It is also sometimes referred to as a Spark *job*. Every `SparkApplication` has a Spark Driver Pod. It runs the process that executes the `main()` method of a `SparkApplication`. It carries out several functions throughout the lifetime of a Spark job:

- The driver initializes the `SparkContext` class, which is the entry point for any Spark functionality.
- It communicates with the Kubernetes API to create executor Pods.
- It coordinates the execution of Spark jobs by breaking it down into tasks.
- It distributes tasks to executors and monitors their execution.
- It collects the results of the tasks executed by the executors.
- It hosts the Spark web UI, typically on port 4040, which provides information about the application's execution.

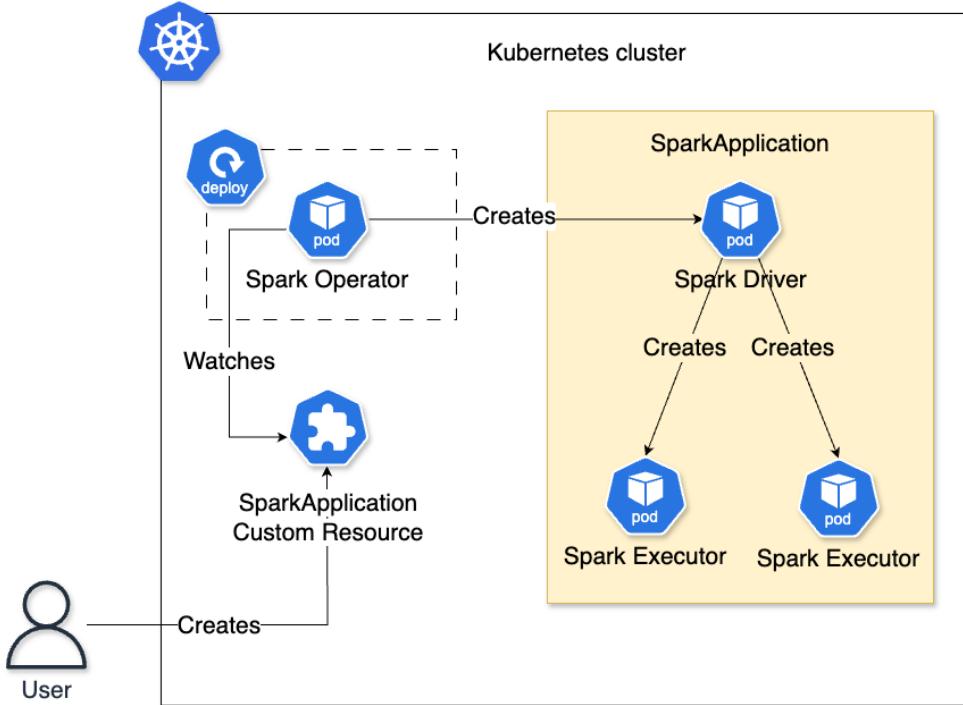


Figure 4.10 Spark components on Kubernetes. The Spark operator watches for `SparkApplication` custom resource. When a new `SparkApplication` resource appears, the operator creates Spark driver Pods, which in turn creates executor Pods and monitors them.

The executor Pods run the processing tasks as directed by the Spark driver. The `SparkApplication` shown in listing 4.18 currently has one executor Pod. Changing the value of `spec.executor.instances` to two will result in the driver creating two executor Pods.

Another benefit of using `SparkOperator` is that it gives you the flexibility to control the properties of the Pod it creates. Do you remember how in Airflow you can use Pod template files to add Persistent Volumes and Config Maps? With `SparkOperator`, you can declare those resources within the `SparkApplication` (see: <https://github.com/kubeflow/spark-operator/blob/master/examples/spark-pi-pod-template.yaml>).

We have set the `deleteOnTermination` flag to `False` for executor Pods, which means once the Spark application completes, Kubernetes will not delete

the driver and executor Pods. You'd usually do this when you need to access logs or other artifacts from a container before the Pods gets deleted. In production, terminated Pods are not meant to linger.

4.12.2 Creating Airflow DAGs for Spark Job

SparkOperator makes it easy to operate Spark clusters in Kubernetes. Integrating the operator with Airflow gives you the ability to use Spark in your data pipelines. `SparkKubernetesOperator` is an Airflow operator that you can use to schedule Spark applications on Kubernetes.

The operator takes a `SparkApplication` definition file in YAML format (as shown in listing 4.19) and submits it to Kubernetes. Then SparkOperator creates Spark driver and executor Pods.

Listing 4.19 A DAG with a Spark application

```
from airflow import DAG
from
airflow.providers.cncf.kubernetes.operators.spark_kubernetes
import SparkKubernetesOperator
from airflow.utils.dates import days_ago

with DAG(
    dag_id="spark_pi_submit_v1",
    start_date=days_ago(1),
    schedule="@daily",
    catchup=False,
):
    submit = SparkKubernetesOperator(
        task_id='spark_pi_submit',
        namespace="airflow",
        application_file='example_spark_kubernetes_operator_pi.yaml',
        do_xcom_push=False,
    )
```

Before you schedule this DAG in your cluster, you must give Airflow additional permissions. Note that the `SparkApplication` you created in listing 4.18 uses the `so-spark` service account. This service account is allowed to create resources in the `spark-operator` namespace. Whereas, Airflow creates resources in the `airflow` namespace. Therefore, we must use the `airflow-worker` service account, which is meant for executor Pods.

What are service accounts?

Service accounts are a part of the Kubernetes authentication and authorization system. They provide identities for processes running in Pods. Why do processes need identities?

Sometimes applications need to communicate with the Kubernetes API to perform an action. For example, a process running a Pod may need get a list of all running Pods in a namespace. By default, Kubernetes will deny such an action. This is because the default service account in a namespace doesn't have the permissions to do so. We can remediate this situation by creating a new service account and adding it to the Pod's configuration.

Then, we can create a `Role` that allows this action (list all Pods). Roles define a set of permissions for listing or changing Kubernetes resources.

Finally, we attach the role to the service account by creating a role binding. Role bindings connect Kubernetes subjects (users, groups, or service accounts) to roles, granting them the permissions defined in the role.

Roles are namespace specific. When you need a role that applies to multiple namespaces, you'll need a `ClusterRole`, which are cluster-wide, applying across all namespaces in the cluster.

A `ClusterRoleBinding` binds a subject to a `ClusterRole`.

This airflow-workers service account can create resources required to run Airflow resources, but it doesn't have permissions to create a `SparkApplication` custom resource. The book's GitHub repository contains a script to add additional permissions to the airflow-worker service account.

```
$ sh add_perms_airflow_worker.sh
```

The last thing you need before you run the DAG is the YAML file that contains the definition of the `SparkApplication` resource. Listing 4.20 shows how you can use not just Python, but also Scala (also Java and R) to write data analysis tasks in Spark.

Save the DAG and the YAML file to the Airflow shared folder and Airflow will trigger the DAG. Here's the sequence of events that occurs when the DAG runs:

1. Airflow schedules the DAG
2. `SparkKubernetesOperator` creates a `SparkApplication` custom resource
3. `SparkOperator` takes record of the new `SparkApplication` and creates Spark driver Pods
4. The Spark driver creates executors and monitors them until they finish

Listing 4.20 Spark application for calculating the value of Pi using Scala

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: spark-pi
spec:
  type: Scala
  mode: cluster
  image: "spark:3.5.1"
  imagePullPolicy: Always
  mainClass: org.apache.spark.examples.SparkPi
  mainApplicationFile: "local:///opt/spark/examples/jars/spark-
examples_2.12-3.5.1.jar"
  sparkVersion: "3.5.1"
  restartPolicy:
    type: Never
  driver:
    cores: 1
    coreLimit: "1200m"
    memory: "512m"
    labels:
      version: 3.5.1
      serviceAccount: airflow-worker
  executor:
    cores: 1
    instances: 2
    memory: "512m"
    labels:
      version: 3.5.1
```

With the addition of Spark, our environment now boasts the capability to host scalable distributed data pipelines. As usage grows and the number and complexity of pipelines increase, Kubernetes performs the seamlessly scales the underlying infrastructure to meet the demands. When data pipelines require file storage, Kubernetes automatically integrates with the backend storage service, provisioning volumes as needed. It scales nodes dynamically to match workload requirements, eliminating the need for manual server. With Kubernetes, we can focus on building our data pipelines, while the platform handles the underlying infrastructure requirements.

4.13 Architected for scale

Highly parallel batch workloads put significant strain on systems. For example, consider a workflow with jobs that are distributed across hundreds of Pods. Whenever this workflow gets triggered, Airflow creates hundreds of new Pods. If this workflow overlaps with others, Airflow creates still more Pods. However, Kubernetes has rate limits to prevent the API server and other resources from getting overwhelmed.

Such a storm has the potential of slowing down the Kubernetes API server. Setting resource limits using Airflow pools should control the concurrency. This way, Airflow can throttle the number of concurrent tasks without slowing down the API server or getting rate limited by the Kubernetes API server.

Creating hundreds of short-lived Pods also increases the churn rate, which is a measure of the rate at which resources like Pods are created and destroyed in a Kubernetes cluster over a given period. A high Pod churn rate not only puts pressure on the Kubernetes API and the kubelet but also any systems that support the tasks.

COREDNS

CoreDNS, which provides name resolution in a Kubernetes cluster, can get overwhelmed when thousands of Pods send requests. This scenario is common in data pipelines and other batch workloads at enterprise-scale.

To address this situation, CoreDNS can be tuned to improve the reliability of the name resolution service within a cluster. Kubernetes documentation (https://github.com/coredns/deployment/blob/master/kubernetes/Scaling_Core_DNS.md) includes strategies for scaling CoreDNS.

It is also a common practice to run a local DNS server on every Kubernetes worker node in larger clusters. *NodeLocal_DNSCache* is a Kubernetes addon that runs a “dnsCache” pod as a DaemonSet to improve DNS performances and reliability.

PICKING THE RIGHT EXECUTOR

A key benefit of the Kubernetes executor is that each task is isolated into a Pod.

The flip side is that every task must create a new Pod. In a busy cluster, creating Pods can be slow, especially for short-lived tasks. Kubernetes is usually very responsive and new pods get scheduled and start running shortly after creation. In a busier cluster with high Pod churn rate, creating new containers may not be instantaneous.

One strategy to keep the Pod churn rate low in environments with many short-lived tasks is to have dedicated Airflow Celery workers. In this setup, Celery workers run as long running Pods. When a DAG starts, the Airflow scheduler runs tasks on one of the Celery worker Pods instead of creating a new Pod to execute the task.

This is made possible by `CeleryKubernetesExecutor`, which is an Airflow executor that lets you pick how you’d like a task to be executed. For example, short-lived may be configured to run on Celery workers and tasks that need isolation run in dedicated Pods.

SCALING AIRFLOW COMPONENTS

For a system to be reliable, it must not have any single points of failure. In the current configuration, the Airflow webserver and scheduler are single points of failure, which may be okay for non-production environments, but in production, these components should be scaled horizontally to make them highly available. Similarly, Airflow's database should be highly available and fault-tolerant.

Another way of making systems more resilient is by running replicas across multiple fault domains. A fault domain can be an Availability Zone (AZ) in the cloud or a datacenter. The idea is to improve the availability of a workload even when an entire Availability Zone or a datacenter goes down.

Kubernetes makes it easier to spread replicas of a workload across multiple nodes and fault domains. This is accomplished by using Pod anti-affinity and topology spread constraints to ensure that replicas of the same component (for example Airflow scheduler) are scheduled on nodes in different AZs.

Summary

- Data pipelines provide a reliable, observable, and consistent way to automate data engineering steps.
- Exploratory Data Analysis (EDA) is the process of analyzing and investigating data sets to summarize their main characteristics, often using visual methods.
- Transitioning from notebooks to pipelines requires breaking down processes into discrete, independent tasks.
- Airflow on Kubernetes utilizes infrastructure resources more efficiently than traditional server setups by creating workers only when needed.
- Airflow runs tasks in workers. In Kubernetes, these workers are Pods whose properties you can control at a DAG or global level.
- XComs allow small amounts of data to be passed between tasks but have size limitations. For larger data transfers, alternatives like custom XCom backends or shared filesystems/object stores are recommended.
- XComs are ideal for passing small amounts of data between tasks. Use external storage for passing processed data to downstream tasks.
- Use KubernetesPodOperator when a task needs a custom container image or when it is written in a programming language other than Python.
- The Spark Operator simplifies deployment and management of Spark applications on Kubernetes.

- Apache Spark can be integrated with Airflow for distributed data processing using `SparkKubernetesOperator`.
- Configuring requests and limits ensures predictable cluster performance and prevents resource contention.
- Implement Kubernetes Resource Quotas to cap aggregate resource consumption per namespace, ensuring fair allocation in shared environments.

5

Training machine learning models on Kubernetes

This chapter covers

- Understanding distributed machine learning training
- Distributed training with TensorFlow and Kubeflow
- Distributed training with PyTorch and Kubeflow
- Using alternate schedulers with Kubernetes

In Chapter 4, you learned how to use distributed architectures to speed up data analytics. It's now time to explore how Kubernetes helps you scale machine learning, specifically, model training.

But first we must answer why you should consider Kubernetes when training models. Of all steps in the ML development lifecycle, model training is the most resource intensive process. Classical ML models like simple classification and regression models, the kinds that were the rage before Transformers came into being, needed far fewer resources than today's multimodal foundation models. Since these classical models were trained on a much smaller volume of data, training them didn't take long.

Newer model architectures now enable us to train models that can handle vast amounts of data and complex tasks, such as natural language processing, image recognition, and multimodal learning. These models, like Transformers and their variants, require massive computational resources to process large datasets efficiently. Training such models involves significant computational power, often necessitating distributed computing environments to handle the scale and complexity of the data. Whereas smaller classical ML models can be trained on a single device (CPU or GPU), modern models demand a distributed setup to achieve reasonable training times.

Training (or fine-tuning) models like Large Language Models and foundational models usually requires a distributed training architecture. The amount of data used to train these models is so big that it would take years to train an AI model like Meta's Llama with a single computer.

Kubernetes is particularly well-suited for managing distributed training at scale. Especially, since there's several open source and commercial products that extend Kubernetes API to add support for distributing training jobs across multiple worker nodes while leveraging GPUs and other specialized hardware to accelerate computation. Besides providing the raw compute, storage, and network, Kubernetes also simplifies:

- Handle infrastructure provisioning and decommissioning – Kubernetes worker nodes autoscale as required. This means that when training demands increase, additional nodes can be dynamically added to handle the workload. Conversely, when tasks are completed, nodes can be automatically decommissioned, reducing costs and optimizing resource utilization.
- Resource allocation – Kubernetes allows for fine-grained control over resource allocation, ensuring that each training task receives the necessary compute resources without over-provisioning. This is particularly important for GPU-intensive workloads, where efficient allocation can significantly impact training times and costs.
- Fault tolerance and reliability - Kubernetes' self-healing capabilities ensure that if any worker node fails during training, the process can be automatically resumed on another available node. This minimizes downtime and ensures that model training remains uninterrupted, which is indispensable in production environments where model updates need to be frequent and reliable.
- Data access security – Kubernetes provides mechanisms that can be used to provide secure access to secrets and confidential data.
- Customizable scheduling – Kubernetes supports bring-your-own scheduler model, which enables advanced scheduling techniques designed to minimize resource wastage.

In addition, Kubernetes clusters can provide cluster-level observability and security policy enforcement solution, freeing ML engineers' time to focus on model development by automating monitoring and security tasks. Having robust observability and centrally-enforced security policies enables proactive detection of issues, ensuring that ML workloads run smoothly and securely, while also enhancing compliance with regulatory standards through detailed logging and auditing capabilities.

Libraries like PyTorch, TensorFlow, and XGBoost provide parallelization techniques to speed up the model training process. They offer built-in support for distributing computations across multiple GPUs and nodes. However, they

require a very specific infrastructure setup, which ML engineers find cumbersome and complex to manage.

In this chapter, you'll learn how Kubeflow, an open-source machine learning platform, allows you to bring the power of distributed computing to model training. Kubeflow provides an abstraction layer that helps ML users run distributed model training and fine tuning jobs without getting lost in the dark and complex world of virtual machines, disks, and subnets. By providing a simple and intuitive interface, it enables you to easily scale your model training jobs to large clusters of machines, making it possible to train complex models in a fraction of the time it would take on a single machine.

In this chapter, we will begin by exploring the foundational techniques of distributed training and gradually delve into their practical application with Kubeflow. To illustrate these concepts, we will use the MNIST dataset to train a Convolutional Neural Network (CNN) for image classification. This model will be trained across a cluster of machines, a process known as distributed training. By the end of this chapter, you will have a clear understanding of how to execute distributed training jobs with TensorFlow and PyTorch on Kubernetes using Kubeflow, equipping you with the skills to scale machine learning workloads efficiently.

5.1 *Distributed training in a nutshell*

The concept of distributed computing goes all the way back to the 1980s when researchers began exploring ways to parallelize neural network training across multiple processors. This exploration was driven by the need to handle larger models and datasets, which were becoming increasingly complex.

In the early 2000s, big data accelerated the need for distributed systems. Google's introduction of the MapReduce in 2004 and the subsequent development of Apache Hadoop provided a framework for distributed processing of large datasets. These technologies provided a framework for distributed processing of large datasets, laying the groundwork for future advancements in distributed computing, even though they were not specifically designed for machine learning tasks.

The 2010s marked a significant turning point with the introduction of tools like Spark MLlib, TensorFlow, and PyTorch. These tools democratized distributed training, making it more accessible to machine learning practitioners. They allowed for the efficient training of models by distributing computations across multiple nodes and GPUs. Since 2020, the development of large language models, such as OpenAI's GPT-3, has pushed the boundaries of distributed training. These models, with hundreds of billions of parameters, require

sophisticated techniques and extensive computational resources, often involving thousands of GPUs, to train effectively.

Large language models have benefited hugely from distributed training. For example, according to the popular model Falcon-40B's documentation, it was trained using 384 Nvidia A100 40GB GPUs. Yet it still took two months to complete the training. It would be impractical to train such models on a single machine.

The fundamental difference between single machine and distributed training is that in distributed training, the training script runs on multiple machines. Each worker (GPU or an entire machine) calculates gradients independently based on its portion of data. The gradients are then periodically aggregated to update the global model.

There are two prevalent strategies for parallelizing model training:

- Data parallelism - Distributes data across multiple nodes, with each node processing a subset of the data.
- Model parallelism - Splits the model across multiple devices, useful for very large models.

We'll focus on data parallelism as it is the most used strategy and is sometimes used in combination with model parallelism.

In data parallelism, the dataset is split into smaller mini-batches and each worker receives a different mini-batch. This execution model is called *Single Program Multiple Data* or SPMD in the High Performance Computing (HPC) world. It implies that all workers execute the same training code but operate on different portions of the input dataset.

After workers process mini-batches, the gradients are aggregated to update the global model. This process is called *gradient aggregation*. Its purpose is to merge the learning from different subsets of data processed by each worker into a single, unified update for the global model.

Forward and backward pass

A neural network consists of layers of connected units or interconnected neurons. The input layer is the entry point of a neural network. It is the first layer in the network and is receives input data from the outside world. This layer does not perform any computations but simply passes the input data to the next layer, typically a hidden layer, where the actual processing begins.

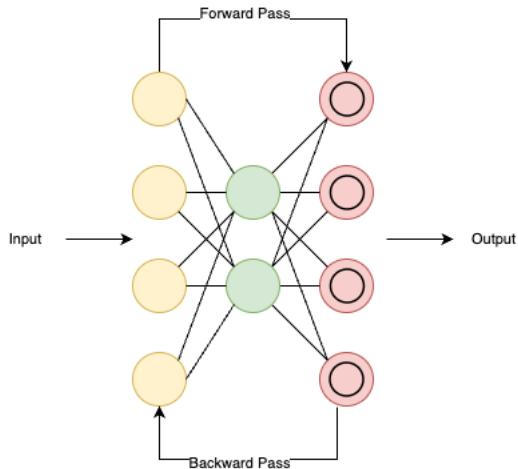


Figure 5.1 A simple neural network architecture, highlighting the forward and backward pass processes. The leftmost layer where data enters the network. The middle layer performs computations. And the rightmost layer produces the network's output.

In traditional feed-forward networks, the data flows in one direction, from the input layer, through one or more hidden layers, to the output layer. Each neuron in the hidden and output layers performs computations by applying weights to the inputs, adding a bias, and then passing the result through an activation function. This is called a *forward pass*. The input data propagates through the network to produce an output. The network then calculates error (or loss) by getting the difference between the prediction and the target value.

Modern neural networks are trained using the backpropagation method. During this phase the error propagates from the output layer back to the input layer, updating the network's weights to minimize the error. The network calculates how much each connection (weight) contributed to the error.

The network adjusts the weights of the connections to reduce the error. This adjustment is done using a method called *gradient descent*, which finds the direction and magnitude of the change needed to minimize the error.

This process is repeated for many iterations (epochs) and for many input examples. Over time, the network learns to make more accurate predictions by continuously adjusting the weights.

The aggregation process is sensitive to network latency and throughput, and it can become a bottleneck in distributed training. This is especially true when dealing with tens or even hundreds of workers. As the number of workers increases, the communication overhead associated with aggregating gradients can slow down the training process. Much research has gone into developing approaches and algorithms that accelerate gradient aggregation.

Two commonly used methods to enable gradient aggregation are:

- Parameter server
- Collective communication

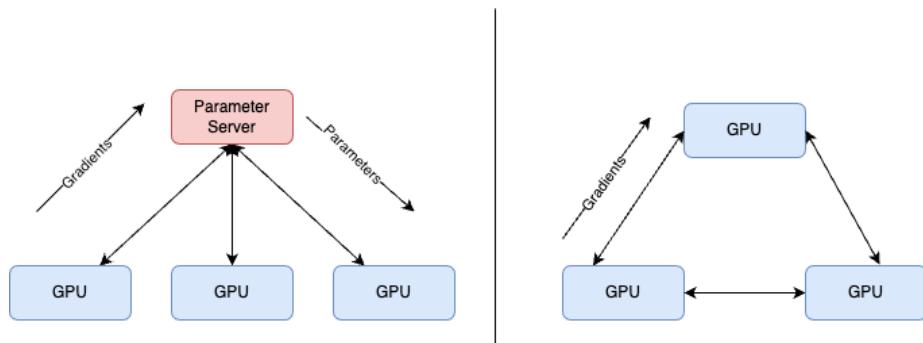


Figure 5.2 Illustration of parameter server (left) and all-reduce (right) architecture. In the parameter server pattern, the workers pull parameters from and push gradients to the parameter server.

When using a parameter server, workers send results to a dedicated server. This server is responsible for storing and updating model parameters. This pattern enables asynchronous training, which is useful in heterogeneous environments where workers may have different processing speeds. Each worker processes data on its own and doesn't depend on other workers. Should a worker fail, the training continues. The key limitation with this approach is that the parameter server itself becomes a bottleneck, especially when there are many workers reading and updating parameters at a rapid pace.

Collective communication, particularly the all-reduce approach, involves direct communication between workers to aggregate gradients. This is a synchronous training pattern. All workers need to be online at all times. If one of them becomes unavailable, the training process halts. In this method, each worker exchanges gradients with other workers in a structured manner to perform the aggregation.

Collective communication pattern is more efficient than the parameter server approach, as it reduces the communication bottleneck by distributing the aggregation workload among all workers. Since it avoids a single point of congestion, it also scales better for large-scale systems. However, the complex data synchronization and coordination among workers requires high bandwidth connectivity between worker nodes. It also requires that all worker nodes be online simultaneously.

Both these strategies have their strengths and weaknesses. Parameter server pattern is suitable when workers have varying compute resources, or their reliability is questionable. Although scaling it to go beyond a few workers might be challenging without running into network bottlenecks.

Even in cloud, the parameter server pattern has its uses. For example, if you're using virtual machines that can be preempted, like Amazon EC2 Spot Instances, and Spot Virtual Machine in Google Cloud and Azure.

Infrastructure reliability is high in cloud environments. This opens the door to more performant and scalable patterns that employ collective communications for information exchange. If your calculations show that there's a strong chance workers will remain online, prefer the collective communications pattern.

5.2 *Distributed training with TensorFlow*

Let's explore the TensorFlow APIs that simplify distributed training. Once you understand how to perform distributed training, you'll also know how single node training works.

TensorFlow provides two notable strategies for multi-node distributed training:

- `MultiWorkerMirroredStrategy` – Uses collective communication (all-reduce algorithm) for aggregating gradients
- `ParameterServerStrategy` – Uses a parameter store for gradient aggregation

Both strategies employ the data parallelism approach.

`tf.distribute.MultiWorkerMirroredStrategy` is an API that runs TensorFlow training across multiple workers, with each potentially running on a different machine. It replicates training variables (like weights, parameters, and gradients) across each worker using collective communications. It requires all workers to be processing simultaneously, which makes it a strategy for synchronous training. If one worker becomes unavailable, other workers must stop their tasks until it returns.

The `ParameterServerStrategy` API supports asynchronous training. Worker process data independently and consolidate gradients using a parameter server.

5.2.1 TensorFlow cluster

In TensorFlow, distributed training occurs in a cluster. Each cluster has “jobs”, and each job can have one or more “tasks”. Depending on the strategy, a cluster has:

- One or more worker jobs. Their task is to process data.
- One or more parameter servers. Their task is to act as the parameter store.
- One coordinator job, called *chief*. It’s responsible for driving the training process.

Usually, the first worker in the cluster is designated as the chief. It handles tasks like writing logs and saving checkpoints. This worker is also sometimes referred to as the coordinator or scheduler in TensorFlow documentation.

5.2.2 MultiWorkerMirroredStrategy

`MultiWorkerMirroredStrategy` is a strategy in which all workers perform training steps in sync. They use peer-to-peer networking (using collective communications) to update model parameters consistently across all devices.

When training starts, the training script replicates all model variables across each device. Each worker maintains a copy of the model, and gradients are aggregated across all workers before updating the model parameters. This may sound a lot but most of these functions are automatically performed by TensorFlow and abstracted for ease of use. Since each worker trains the model on its own mini-batches, TensorFlow provides abstractions to handle batching, sharding, and prefetching of data.

A distributed training script is very much like a single device training script. Besides parallel training, the key differences are:

3. The variables for model layers are copied to all workers
4. Each worker trains on a non-overlapping subset of data
5. At the end of each batch, workers aggregate gradients using all-reduce algorithm
6. Workers get a new batch of data and repeat the process 2 and 3
7. The chief creates checkpoints and eventually saves the model to disk

Converting a single node training script to run across multiple nodes is not a trivial task. However, it is not monumental either. Arguably, the more challenging

part is setting up the cluster and ensuring workers don't waste a lot of time aggregating gradients.

One problem with distributed training is that workers spend a significant amount of time waiting for the variables to be updated. To reduce the time it takes to update parameters, workers need low latency, high throughput network. Hardware solutions like InfiniBand, RDMA over Converged Ethernet (RoCE), and Amazon Elastic Fabric Adapter provide specialized networking, which are adept for speeding up distributed training. Depending on your environment, you may have to install device drivers in your cluster and include communications libraries in the container images. Please consult your provider's documentation for the specifics.

Let's turn our focus to `ParameterServerStrategy`, which is very similar to `MultiWorkerMirroredStrategy`, with a few differences.

5.2.3 ParameterServerStrategy

A parameter server training cluster consists of a coordinator (chief) with one or more workers and parameter servers. At the start of the training, the chief creates variables on parameter servers and defines the functions workers will execute.

The key benefit of this strategy is that it allows the training process to be more fault tolerant. Unlike in `MultiWorkerMirroredStrategy`, training continues should one or more workers become unavailable.

In this strategy, the workers are stateless, and parameter servers maintain the current version of a model's variables. The coordinator dispatches work to workers. Workers process mini-batches by performing these steps:

1. The workers read initial set of variables from the parameter server
2. They read input data (mini-batch)
3. They compute the loss function (forward pass)
4. They compute the gradients (backward pass)
5. They write the gradients back to the parameter store

This process repeats until the end of training.

5.3 Training models with Kubeflow

As we discussed in the previous sections, distributed training requires setting up a cluster, which can be a complex and time-consuming process. Kubeflow makes it easy to set up and manage distributed training clusters.

Kubeflow is an open-source machine learning and MLOps platform built on Kubernetes. It helps organizations build, deploy, and manage ML workflows and models.

Kubeflow comes in two flavors: Kubeflow Platform and standalone tools. The Kubeflow Platform is a suite of ML tools bundled together with additional integration and management tools. It includes:

- Kubeflow Trainer – For training and fine tuning models
- Kubeflow Pipelines - Kubeflow allows users to define and execute ML pipelines, which are a series of tasks that are executed in a specific order to train, test, and deploy ML models.
- Kubeflow Notebooks – JupyterHub for model development (like the JupyterHub environment in your cluster).
- Kubeflow Katib – For hyperparameter tuning and model optimization
- Kubeflow Model Registry – For storing store ML metadata, model artifacts, and preparing models for production serving.
- Kubeflow KServe – For online and batch inference.
- Kubeflow Spark Operator – For running Spark jobs (we discussed this in Chapter 4).

The Kubeflow Platform is pre-configured and pre-integrated distribution of these tools. In addition, Kubeflow provides functions such as user management (it uses Dex instead of Keycloak). Together these provide end-to-end machine learning capabilities. The Kubeflow Platform is designed to be cloud-agnostic, meaning it can be deployed on multiple cloud providers, including Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure, and others. However, the level of support and integration with each cloud provider can vary.

Instead of using the Kubeflow Platform, we'll use its standalone components in this book. By using the standalone components of Kubeflow, you can avoid the complexity of setting up the Kubeflow Platform, which requires significant resources and expertise. The standalone components provide a more flexible and customizable way to use Kubeflow, allowing you to choose the components that you need and integrate them with your other tools and services in your environment. For example, instead of using Kubeflow Notebooks to get Jupyter notebooks, we used JupyterHub in this book.

Indeed, you are already familiar with one of the Kubeflow components: the Spark Operator. Just as we used Spark Operator without installing other Kubeflow components, we'll now show you how to use the Kubeflow Trainer to run a distributed training job in our Kubernetes cluster.

5.3.1 *Installing Kubeflow Trainer*

Kubeflow Trainer (formerly known as Kubeflow Training Operator) is designed to simplify distributed model training. It enables scalable, distributed training across various frameworks, including PyTorch, JAX, TensorFlow, and XGBoost.

To install Kubeflow Training, first create a namespace:

```
$ kubectl create ns kubeflow
```

Install the Training Operator using kubectl:

```
$ kubectl apply -k "github.com/kubeflow/training-
operator/manifests/overlays/standalone?ref=v1.7.0"
```

You should now have a Training Operator Pod running in the kubeflow namespace.

```
$ kubectl get pods -n kubeflow
NAME                               READY
STATUS      RESTARTS   AGE
training-operator-64c768746c-9pf9q   1/1
Running          0           2h
```

The operator also creates CRDs in the cluster. These CRDs support distributed training for various frameworks.

Remember how you created a `SparkApplication` CRD to run a Spark job in Kubernetes? To create a TensorFlow training cluster, you'll create a `TFJob` resource. The Training Operator supports the following frameworks:

- `TFJob` for TensorFlow
- `PyTorchJob` for PyTorch
- `MPIJob` for Message Passing Interface (MPI)
- `MXNetJob` for Apache MXNet (to be deprecated in future Kubeflow versions)
- `XGBoostJob` for XGBoost

```
$ kubectl get crds | grep kubeflow
mpijobs.kubeflow.org
mxjobs.kubeflow.org
paddlejobs.kubeflow.org
pytorchjobs.kubeflow.org
tfjobs.kubeflow.org
xgboostjobs.kubeflow.org
```

We'll discuss `TFJob`, `PyTorchJob`, and `MPIJob` in this book. Since we began this chapter with TensorFlow, let's review `TFJob` first.

5.3.2 *Training TensorFlow models with Kubeflow*

A `TFJob` is a custom resource that defines how a TensorFlow training cluster should be setup and managed. It includes the configuration of the training job, workers, parameter servers, chiefs, and container image to use for training job.

Listing 5.1 A sample `TFJob` manifest

```

apiVersion: kubeflow.org/v1 #A
kind: TFJob #B
metadata:
  generateName: tfjob
  namespace: your-user-namespace
spec:
  runPolicy:
    cleanPodPolicy: Running #C
  tfReplicaSpecs:
    PS: #D
      replicas: 1 #E
      restartPolicy: OnFailure
      template:
        metadata:
          annotations:
            sidecar.istio.io/inject: "false" #F
        spec:
          containers:
            - name: tensorflow
              image: your-image
              command:
                - python
                - training-script.py #G
                - --batch_size=32
    Worker:
      replicas: 3 #H
      restartPolicy: OnFailure
      template:
        metadata:
          annotations:
            sidecar.istio.io/inject: "false"
        spec:
          containers:
            - name: tensorflow
              image: your-image
              command:
                - python
                - training-script.py
                - --batch_size=32

```

#A Indicates that the resource is using the Kubeflow API version 1.0.

#B The resource being defined is a TFJob

#C Controls Pod cleanup behavior after job completion. Running means only pods still running when a job completes (e.g. parameter servers) will be deleted immediately; completed pods will not be deleted so that the logs will be preserved. This differs from All (delete all pods) or None (no cleanup).

#D The parameter server section

#E Number of desired parameter server replicas

#F Required when using Kubeflow Trainer as a standalone operator

#G Training script

#H The desired count of workers

As you might have guessed by the fact that there's a parameter section defined, the Python script that runs in the TFJob shown in listing 5.1 uses the parameter server strategy. Notice that the parameter server and worker containers setup is

identical. Both containers run the same script with the same parameters. When scripts run, they determine their role and assume responsibilities. Using the same or different scripts for parameter server and workers is a personal preference.

The parameter section will be absent for training scripts that use the MultiWorkerMirroredStrategy. The book’s GitHub repository contains a model training script and a TFJob resource YAML file that uses the MultiWorkerMirroredStrategy (Chapter 5/kubeflow/TensorFlow/multi_worker_strategy-with-keras.py).

The authors have built a custom container image using an Nvidia-provided TensorFlow image. The training script for a residual neural network (ResNet) model has been added to the base image (Chapter 5/kubeflow/TensorFlow/multi_worker_strategy-with-keras.py). The model is trained using the MNIST dataset, which consists of 70,000 small, 28x28 grayscale images of handwritten digits (0-9). We’ll train the model to predict the correct digit label for each image. You can find the Dockerfile for this container in the book’s repository (Chapter 5/kubeflow/TensorFlow/Dockerfile).

Training starts as soon as we submit the TFJob YAML to Kubernetes and there are nodes available to run the job. Listing 5.2 shows a TFJob manifest for MultiWorkerMirroredStrategy. This training cluster will have two workers corresponding to two Pods. Each worker requests one NVIDIA GPU. If there are no nodes with GPU available in the cluster, the cluster autoscaler (Karpenter in this case), provisions worker nodes with one or more GPUs. The job needs at least two GPUs, so we need one worker node with two or more GPUs or two nodes with a GPU.

Listing 5.2 A TFJob with MultiWorkerMirroredStrategy

```
apiVersion: kubeflow.org/v1
kind: TFJob
metadata:
  name: multi-worker
spec:
  runPolicy:
    cleanPodPolicy: Running
  tfReplicaSpecs:
    Worker:
      replicas: 2 #A
      restartPolicy: Never #B
      template:
        spec:
          containers:
            - name: tensorflow
              image: realz/kubeflow-tfjob:v1 #C
              volumeMounts:
                - mountPath: /train
                  name: training
```

```

resources:
  limits:
    nvidia.com/gpu: 1
volumes: #D
- name: training
  persistentVolumeClaim:
    claimName: kubeflow-efs-shared
#A These workers will process different data partitions while sharing model
parameters
#B Prevents automatic restarts of failed worker pods. This is stricter than OnFailure
and forces manual intervention for failures, useful for debugging hard crashes
#C Custom Docker image containing TensorFlow training code and dependencies.
Must include distributed training logic in the Python script (e.g., using
tf.distribute.MultiWorkerMirroredStrategy).
#D Mounts a Persistent Volume Claim at 'train' for shared storage. Used for store
training data/checkpoints
#E Requests 1 GPU per worker Pod

```

When this container starts, it runs `multi_worker_strategy-with-keras.py`. This script saves the model to the local filesystem at the end of training. We add a Persistent Volume to the job, so the model artifacts are saved on the shared file system.

Since we haven't specified a namespace, this training job runs on the default namespace. To create a Persistent Volume and Persistent Volume Claim, run the script in 'Chapter 5/kubeflow':

```
$ cd "Chapter 5/kubeflow"
$ sh create-efs-pv.sh
```

CAUTION. The next step creates a distributed training job using two GPUs.

Please understand the cost implications before proceeding. For example, in AWS, getting an EC2 instance with two GPUs (or getting two with one GPU each) costs roughly \$5 an hour. It takes about 5-10 minutes to complete the training job, which costs about a dollar per run.

To submit the job to Kubernetes, run:

```
$ kubectl apply -f TensorFlow/tfjob-mwms.yaml
```

It can take a few minutes for the job to start. This is because the nodes have to download this huge container image. Container images for machine learning workloads can be significantly larger than images for a web application server like NGINX. The size of this job's container image is at whopping 13 GB.

Once the Pod starts running, we can see the training logs:

```
$ kubectl logs multi-worker-worker-0 -f
...
Learning rate for epoch 4 is 9.999999747378752e-05
70/70 [=====] - 1s 19ms/step - loss: 0.1009 - accuracy: 0.9693 - lr: 1.0000e-04
Epoch 5/10
67/70 [=====>..] - ETA: 0s - loss: 0.2240
```

```
- accuracy: 1.9284
Learning rate for epoch 5 is 9.999999747378752e-05
70/70 [=====] - 1s 19ms/step - loss:
0.1116 - accuracy: 0.9641 - lr: 1.0000e-04
Epoch 6/10
68/70 [=====>.] - ETA: 0s - loss: 0.2078
- accuracy: 1.9375
...
Epoch 8/10
70/70 [=====] - ETA: 0s - loss: 0.1725
- accuracy: 1.9480
Learning rate for epoch 8 is 9.999999747378752e-06
70/70 [=====] - 1s 16ms/step - loss:
0.0862 - accuracy: 0.9740 - lr: 1.0000e-05
Epoch 9/10
...
Epoch 10/10
69/70 [=====>.] - ETA: 0s - loss: 0.1746
- accuracy: 1.9488
Learning rate for epoch 10 is 9.999999747378752e-06
70/70 [=====] - 1s 16ms/step - loss:
0.0868 - accuracy: 0.9746 - lr: 1.0000e-05
```

Notice that in listing 5.2, we've set `spec.runPolicy.cleanPodPolicy` to `Running`, which tells the Kubeflow that we don't want it to terminate the Pods once training ends. We do this so we can view logs. Without this setting, Pods will be deleted as soon as training finishes and the logs will be lost.

After viewing the logs, consider deleting the job as soon as possible to avoid paying for GPU instances. Delete the job by running:

```
$ kubectl delete tfjobs multi-worker
tfjob.kubeflow.org "multi-worker" deleted
```

Where to store the training script?

In the example above, the container image includes the training script. If you follow this technique, you end up with an image for each training script version. Since every time you update the script, you'll must build an image with a newer tag.

```
Training script (v1) -> my_image:v1
Training script (v2) -> my_image:v2
```

This may become costly depending on your architecture. Another way is to fetch the training script at runtime. You can download the image from a code repository or read it from a shared storage. This way you have a generic image, and you specify which script to run at runtime.

5.3.3 Recapping the process

Now you know how to train TensorFlow models on Kubernetes using Kubeflow. Let's recap the process:

Decide between synchronous and asynchronous training. If workers are unreliable, use a parameter store. Otherwise use MultiWorkerMirroredStrategy.

6. Implement the strategy into your training script.
7. Create a container image with the script. Even better, decouple the container image and the training script by storing the training script as a ConfigMap.
8. Create a TFJob custom resource to create a TensorFlow training cluster in Kubernetes.

Can you guess how you'd train a model with a single worker? Just set the worker replicas to 1.

```
spec:  
  runPolicy:  
    cleanPodPolicy: Running  
  tfReplicaSpecs:  
    Worker:  
      replicas: 1
```

MultiWorkerMirroredStrategy is going to be the most scalable because you don't rely on the availability of parameter stores. This strategy also reduces training time as it benefits from specialized hardware for high throughput, low latency network.

Now that you have a firm grasp of distributed training with TensorFlow and Kubeflow, you'll have no difficulty understanding the concepts of PyTorch distributed training.

5.4 Distributed training with PyTorch

Like TensorFlow, PyTorch offers APIs that enable data and model parallelism. There are four parallelism techniques available in PyTorch:

- Distributed Data-Parallel (DDP)
- Fully Sharded Data-Parallel Training (FSDP)
- Tensor Parallel (TP)
- Pipeline Parallel (PP)

DDP and FSDP are based on data parallelism, while TP and PP are methods of model parallelism. Data parallelism has the widest adoption, as PyTorch 2.0 still has experimental support for TP and PP.

For larger models that don't fit on a single GPU, PyTorch supports combining DDP with model parallelism or using FSDP. Here's the key difference between DDP and FSDP: in DDP, each device retains the entire model, whereas in FSDP, the model's parameters are sharded across multiple devices. FSDP is suitable when you have a model that's too large for a single GPU. FSDP shards the model's parameters across data parallel workers and can optionally offload part of the training computation to the CPUs.

Regardless of the parallelism method, distributed training requires a cluster (or a *process group* in PyTorch parlance) of devices or machines that coordinate work using low-level collective communications libraries like NCCL or peer-to-peer networking using distributed remote procedure call (RPC) framework. To support asynchronous training, you can implement a parameter server to consolidate gradients instead of using collective communication (https://pytorch.org/tutorials/intermediate/rpc_param_server_tutorial.html).

The *world size* refers to the total number of processes participating in the training, typically one process per GPU. For example, if you have two machines and each has two GPUs, the world size is 4. Each process is assigned a unique *global rank*, ranging from 0 to world size minus one. Additionally, each process may also have a *local rank*, which identifies its index relative to the other processes on the same machine. In this setup, you run the same training script on all processes, with PyTorch using the rank information to coordinate work across the cluster.

For example, if you have two virtual machines with eight GPUs each, the world size will be sixteen. Each worker (or rank) uses a GPU. The rank of worker-0 using GPU-0 on machine-0 will be 0. Worker-1 on machine-0 using GPU-1 will have rank=1, and so on.

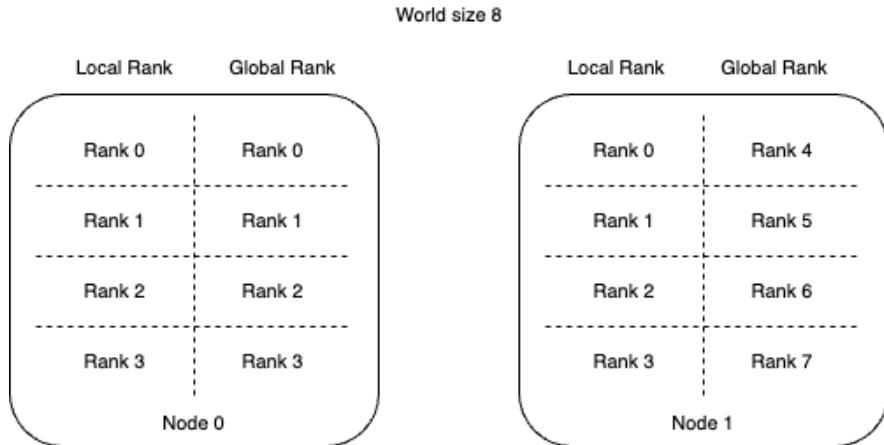


Figure 5.3 A training cluster comprises multiple machines with one or more GPUs. The world size is the number of workers and local rank is the index of worker within the cluster.

Conventionally, the rank-0 worker is designated to be the master node. Conceptually, it is like a TensorFlow chief or coordinator. The master node has added responsibilities for driving the training on workers. For example, in DDP, the rank-0 process broadcasts the initial model state to all the other ranks.

After implementing parallelization, you must start the training script on multiple machines (or devices) using `torchrun` or a similar method. These methods are designed to simplify launching distributed training jobs across a cluster of devices. They handle the setup of multiple processes and their communication. Each training script instance gets a unique local rank, which not only acts as an identifier but is also used in executing worker-specific tasks (for instance, `worker0` is assigned the master).

5.4.1 PyTorch training with Kubeflow

The Kubeflow Training Operator provides the `PyTorchJob` CRD to train PyTorch models on single and multiple nodes. `PyTorchJob` is almost identical to the `TFJob` custom resource we discussed earlier in this chapter.

The operator manages, scales, and cleans up the PyTorch process group (master and workers) on Kubernetes. When you create a `PyTorchJob` resource, this operator creates Pods with appropriate environment variables for the `torchrun` CLI to start distributed PyTorch training job.

`PyTorchJob` has two key components:

- Master – Should only have one replica. It shows the job status, aggregates

model parameters, and create model checkpoints

- Workers – One or many workers that run training (and validation) code.

When a user creates a `PyTorchJob` resource, Kubeflow creates master and worker Pods in which the corresponding training logic gets executed. Kubeflow Trainer configures environment variables such as `WORLD_SIZE` (the total number of workers plus the master) and `RANK` in Pods for identification. Checkout the distributed model training script included in this book's GitHub repository at Chapter 5/kubeflow/PyTorch/mnist_DDP.py. You'll notice that the script only writes checkpoints when `RANK` is 0, that is, the master Pod.

```
if (os.environ["RANK"] == "0"):  
    checkpoint_file = os.path.join(args.dir, "mnist_cnn.pt")
```

When a Kubeflow training job is started, it configures the worker Pods to remain in a waiting state until the master Pod is available. This synchronization is achieved using an Init container, ensuring that training does not begin until both the master and all worker Pods are fully operational. However, this approach can lead to inefficiencies if the cluster lacks sufficient capacity to run all the necessary Pods simultaneously. In such scenarios, Kubernetes may schedule some Pods, resulting in resource allocation and associated costs, even though the training process cannot begin until all Pods are ready.

To address this issue, *gang scheduling* is employed. This scheduling technique ensures that a workload is only executed when all required resources, distributed across various subcomponents, are available simultaneously. For instance, consider a training job that requires four worker Pods and a master Pod, with each Pod needing one GPU. With gang scheduling, the cluster will not start any Pods unless it has at least five GPUs available, preventing partial resource allocation and ensuring efficient utilization. We will explore gang scheduling in more detail later in this chapter.

The job shown in listing 5.3 has a master and worker replica. This job doesn't use any GPUs and runs on CPUs exclusively. Not using GPU also means that the job cannot leverage NVIDIA's NCCL backend for collective communications. For CPU-only training, PyTorch utilizes *Gloo* (<https://github.com/facebookincubator/gloo>), an open source collective communications library by Facebook.

Listing 5.3 A PyTorchJob with a master and a worker

```
apiVersion: "kubeflow.org/v1"  
kind: "PyTorchJob"  
metadata:  
  name: "pytorch-dist-mnist-gloo"  
spec:  
  pytorchReplicaSpecs:  
    Master:
```

```

replicas: 1
restartPolicy: OnFailure
template:
  spec:
    containers:
      - name: pytorch
        image: kubeflow/pytorch-dist-mnist:v1-736c814
        command: ["python", "-u",
          "/opt/mnist/src/mnist_DDP.py"] #A
        args: ["--backend", "gloo", "--dir",
          "/mnt/pytorch/logs", "--save-model"] #B
        volumeMounts:
          - mountPath: /mnt/pytorch #C
            name: training
            subPath: pytorchjobs
          - mountPath: /opt/mnist/src/ #D
            name: pytorchjob-mnist-ddp
    volumes:
      - name: training
        persistentVolumeClaim:
          claimName: kubeflow-efs-shared
          readOnly: false
      - name: pytorchjob-mnist-ddp
        configMap:
          name: pytorchjob-mnist-ddp
Worker:
  replicas: 1 #E
  restartPolicy: OnFailure
  template:
    spec:
      containers:
        - name: pytorch
          image: kubeflow/pytorch-dist-mnist:v1-736c814
          command: ["python", "/opt/mnist/src/mnist_DDP.py"]
          args: ["--backend", "gloo", "--dir",
            "/mnt/pytorch/logs", "--save-model"]
          - mountPath: /opt/mnist/src/
            name: pytorchjob-mnist-ddp
      volumes:
        - name: pytorchjob-mnist-ddp
          configMap:
            name: pytorchjob-mnist-ddp
#A The distributed training script
#B Use gloo backend for CPU-based distributed training
#C Volume mount to save the trained model on a shared filesystem
#D ConfigMap containing the training script
#E Number of workers

```

NOTE. Did you notice how similar PyTorchJob and TFJob definitions are? Kubeflow Trainer operator provides standardized API, helping you can parallelize model training with any of the supported libraries (such as PyTorch, TensorFlow, MXNet, and XGBoost) with minimal changes to the

workflow. This abstraction enables supporting multiple frameworks and simplifies scaling training workloads in Kubernetes environments.

Whenever performing distributed training in CPU clusters, you'll use Gloo as the backend. For production-grade CPU training, Intel's oneCCL (<https://github.com/uxlfoundation/oneCCL>) (via `oneccl_bindings_for_pytorch`) can further optimize collective operations on Xeon processors, achieving near-linear scaling on CPU clusters. However, Gloo remains the default choice for cross-platform CPU training in PyTorch.

PyTorch documentation recommends prioritizing NCCL for GPU-based training and Gloo for CPU-based training, with exceptions based on hardware infrastructure. For GPU clusters, NCCL is the optimal choice regardless of interconnect: it delivers superior performance for both InfiniBand (via GPUDirect support) and Ethernet environments, particularly in multi-node setups. If NCCL encounters compatibility issues, Gloo serves as a reliable fallback, albeit with reduced GPU efficiency.

For CPU clusters, Gloo is the default for Ethernet networks. There are advanced ways that use InfiniBand setups but support for them isn't yet available natively in Gloo. NCCL's GPU-optimized collective operations leverage high-speed interconnects 3–5× faster than CPU-based Gloo, making it indispensable for large-scale model training. Reserve CPU/Gloo for smaller workloads, prototyping, or environments where GPU access is constrained.

To kickoff this PyTorch distributed training script, first we must place the script in a ConfigMap so we can mount it in Pods. Doing so separates application code from container images, allowing for instant script updates without rebuilding Docker images. By storing the script in a ConfigMap, we can easily update the script and restart pods to apply changes, ensuring version-controlled configuration management. Create a ConfigMap from the Python script:

```
$ kubectl create configmap pytorchjob-mnist-ddp --from-file  
mnist_DDP.py  
$ kubectl apply -f pytorch-mnist-ddp-cpu.yaml
```

This creates two Pods: one for master and worker each. Once the master Pod enters "Running" state, you can view training progress by inspecting the Pod's logs:

```
$ kubectl logs -f pytorch-dist-mnist-gloo-master-0
```

The training script saves the model and training logs to the shared NFS volume. Once the training finishes, you can view training logs from JupyterHub (at `/home/shared/pytorchjobs/logs/`). To view logs from JupyterHub, run:

```
$ ls /home/shared/pytorchjobs/logs/  
events.out.tfevents.1741714564.pytorch-dist-mnist-gloo-master-0  
mnist_cnn.pt
```

The .pt file is the saved model. The file that starts with "events.out" contains the training log, which can be used with TensorBoard to visualize metrics such as loss, accuracy, and other performance metrics.

Once you are done exploring, delete the job by running:

```
$ kubectl delete -f pytorch-mnist-ddp-cpu.yaml
```

And, that's how you can train a model defined using PyTorch across a cluster of machines. All you need for distributed training is a dataset, training script, and YAML file that defines how to execute the training job. Kubeflow abstracts infrastructure management, allowing seamless scaling of PyTorch workloads across nodes while optimizing resource utilization and fault tolerance thanks to Kubernetes.

Now, let's take a look at another distributed training method that's more framework-agnostic.

5.5 *Running MPI jobs with Kubeflow*

The *Message Passing Interface* (MPI) is a standardized tool in the High Performance Computing (HPC) field. It provides point-to-point and collective communications, enabling efficient data exchange and synchronization among distributed processes. It is widely used in parallel computing applications where tasks must be executed concurrently across multiple processors or nodes.

The Kubeflow MPIJob CRD lets you run distributed MPI jobs Kubernetes. The MPI Operator, which manages MPIJob resources, facilitates all-reduce-style distributed training, making it easier to scale machine learning tasks across multiple GPUs and nodes.

The MPIJob resource defines launcher and worker pods. The launcher pod initiates the MPI job using mpirun. It is responsible for setting up the MPI environment and coordinating the execution of the job across worker pods.

When do you pick MPI over other distributed training methods? MPI's key advantage is its support for multiple ML frameworks. While TFJob and PyTorchJob only support TensorFlow and PyTorch respectively, MPI is decoupled from the underlying framework. It works with many frameworks including TensorFlow, PyTorch, and Apache MXNet.

Most ML teams pick MPI in conjunction with Horovod to accelerate ML training. *Horovod* (<https://horovod.readthedocs.io>) is a distributed deep learning training framework that supports TensorFlow, Keras, PyTorch, and Apache MXNet. Originally developed by Uber, Horovod makes distributed training fast and easy to use, significantly reducing model training time from days or weeks to hours or minutes. Converting a single-GPU training script into a distributed one requires just a few lines of Horovod-specific code. It extends MPI and NCCL to provide optimized algorithms for sharing data between training processes.

Horovod's key distinction is that it can use ring all-reduce algorithm to efficiently aggregate gradients across all participating GPUs. Ring all-reduce is faster than all-reduce algorithm for gradient aggregation. This algorithm divides the data to be communicated into chunks and distributes them across a ring of GPUs, ensuring efficient data transfer and reducing the bandwidth required for communication between nodes.

For an example of Horovod with MPI and TensorFlow, please see Chapter 5/kubeflow/MPI/tensorflow_mnist.py.

Together, MPI and Horovod allow you to switch between different deep learning frameworks without having to learn new distributed training paradigms.

Listing 5.4 shows a snippet of an MPIJob custom resource that creates a TensorFlow-based training job. It creates a launcher Pod, which sets up the MPI environment and coordinates the execution of the job across two worker Pods.

Listing 5.4 An MPIJob with Horovod (tensorflow-mpi.yaml)

```
apiVersion: kubeflow.org/v1
kind: MPIJob
metadata:
  name: tensorflow-mnist
spec:
  slotsPerWorker: 1 #A
  runPolicy:
    cleanPodPolicy: Running
  mpiReplicaSpecs:
    Launcher:
      replicas: 1
      template:
        spec:
          containers:
            - image: docker.io/kubeflow/mpi-horovod-mnist
              name: mpi-launcher
              command:
                - mpirun
              args:
                ...
                - python
                - /examples/tensorflow_mnist.py
    Worker:
      replicas: 2
      template:
        spec:
          containers:
            - image: docker.io/kubeflow/mpi-horovod-mnist
              name: mpi-worker
```

#A specifies the number of parallel computation slots or threads that each worker process in a distributed training job can utilize

The `slotsPerWorker` parameter specifies the number of slots allocated to each worker in a distributed MPI job. It determines how many MPI processes each worker node will run.

Imagine you have a cluster with nodes that each have 8 GPUs. You want to run an MPI job with 16 processes distributed across 2 nodes. By setting `slotsPerWorker` to 8, you ensure each node will run 8 MPI processes, fully utilizing the GPUs available on each node.

For each MPIJob, the MPI Operator creates a `ConfigMap` that delivers essential files such as `hostfile` and `kubexec.sh`. The `hostfile` contains the list of worker pods, and `kubexec.sh` is used by `mpirun` on the launcher pod to launch processes on worker pods.

5.5.1 Fault tolerant training with Elastic Horovod

In MPI, the launcher process discovers workers at startup. Should a worker become unavailable during training, the training job stops. Elastic is a feature of Horovod that allows you to scale up and down the number of workers dynamically at runtime, without requiring a restart or resuming from checkpoints. With elastic training, workers can come and go from the Horovod job without interrupting the training process.

Elastic Horovod uses a host discovery script to dynamically identify available hosts during training. This script periodically checks for available nodes and adjusts the training job accordingly. For example, if a node becomes unavailable, the script will update the list of hosts, and Elastic Horovod will reconfigure the training job to continue with the remaining nodes.

Elastic training jobs are started using the `horovodrun` command line tool. The major difference when launching elastic jobs is that hosts are not specified explicitly, but instead discovered at runtime. The most general way to allow Horovod to discover available hosts is to provide a `--host-discovery-script` when launching the job.

Listing 5.5 shows an Elastic Horovod MPIJob. If the number of available slots (`replicas`) falls below the minimum (because of preemption or failure), then the job will pause waiting for more hosts to become available or until `HOROVOD_ELASTIC_TIMEOUT` (default: 600 seconds) has elapsed. If unspecified, minimum `replicas` defaults to `-np`, which represents the desired replica count.

Listing 5.5 An Elastic Horovod MPIJob

```
apiVersion: kubeflow.org/v1
kind: MPIJob
metadata:
  name: tensorflow-mnist-elastic
```

```

spec:
  slotsPerWorker: 1
  cleanPodPolicy: Running
  mpiReplicaSpecs:
    Launcher:
      replicas: 1
      template:
        spec:
          containers:
            - image: horovod/horovod:0.20.0-tf2.3.0-torch1.6.0-
mxnet1.5.0-py3.7-cpu
              name: mpi-launcher
              command:
                - horovodrun
              args:
                - --min-np #A
                - "2"
                - --max-np #C
                - "3"
                - --host-discovery-script #D
                - /etc/mpi/discover_hosts.sh
                - python
                - /examples/elastic/tensorflow2_mnist_elastic.py
    Worker:
      ...
      #A The desired count of training processes (set to 2)
      #B The minimum count of training processes (set to 1)
      #C The maximum count of training processes (set to 3)
      #D The host discovery script

```

The parameter `max-np` caps the number of processes (to prevent over-utilizing available resources) and to serve as a reference point for learning rate scales and data partitions (especially when these need to be held constant regardless of the current number of workers).

With Elastic Horovod, a training job keeps running as the underlying infrastructure scales in and out. If a Pod crashes or gets preempted, Kubernetes reschedules the Pod until the job succeeds or the retry limit is reached.

5.6 *Improving efficiency with Alternate Kubernetes schedulers*

If you've been following along with the training examples, you'd have noticed that Kubernetes starts scheduling training jobs as soon as you create a `TFJob`, `PyTorchJob`, or `MPIJob` CRD and capacity is available. In production clusters, this may not be the case. Kubernetes clusters hosting multiple workloads may not always have the capacity to run a job.

Synchronous training requires all workers to be online or the training halts. In production clusters the total capacity is limited. In other words, a cluster can

only add so many virtual machines before running out of capacity or hitting another limit. It is a common practice for platform engineers to limit the number of virtual machines a cluster can scale to at any point. This is usually done to prevent infrastructure costs getting out of control. GPUs are expensive resources. ML teams on limited budgets cap the number of GPUs to keep infrastructure expense at check.

In such environments, data processing and model training jobs compete for limited resources. Maximizing cluster utilization and while limiting the spend on GPUs is difficult.

Consider a cluster that can scale up to 4 nodes. Each node has one GPU. That gives you up to 4 GPUs to run training jobs. Now suppose you start a training job A that requires 3 GPUs. Assuming you have 0 nodes with GPU, Kubernetes autoscaler (Karpenter is used in this book) will create EC2 instances with 1 GPU each. While this job is still running, you trigger another training job B, which requires 2 GPUs. What do you think Kubernetes will do? Well, the cluster isn't at capacity. Karpenter will notice that job B requires 2 Pods with a GPU each. It will create the fourth machine. No more virtual machines will be provisioned as the limit is set to four. When the fourth machine becomes available, Kubernetes will schedule one Pod from job B. But because job B requires at least 2 Pods to start the training, the fourth instance will run a Pod that's just wasting resources while waiting for the second Pod to come online.

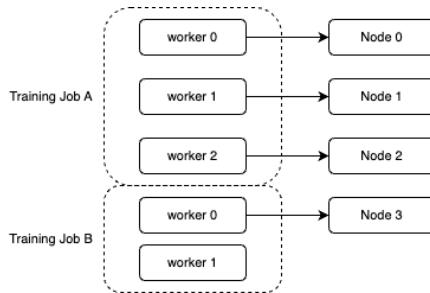


Figure 5.4 With limited compute capacity, a cluster may not have enough resources to run all training Pods simultaneously. When this occurs, Kubernetes may schedule some replicas, which leads to wasted resources as the training cannot begin until all replicas are online.

A better approach will be to have Kubernetes schedule job B when it can guarantee the availability of at least 2 worker nodes with a GPU each. Otherwise, not. This all or nothing scheduling strategy is called *gang scheduling*. Kubernetes, while inherently a single-tenant orchestrator, provides foundational tools for

multi-tenancy, ideal for machine learning teams sharing cluster resources. By combining native Kubernetes constructs like namespaces, RBAC, and resource quotas with advanced schedulers like Volcano or Kueue, organizations can achieve efficient resource isolation and workload prioritization.

By default, gang scheduling isn't possible on Kubernetes. Several custom schedulers like Volcano, Kueue, and YuniKorn enable you to customize scheduling and implement gang scheduling in Kubernetes. Kubeflow also supports these schedulers, giving you the ability to schedule your training runs efficiently. Let's explore this using an example that uses Volcano.

Volcano (<https://volcano.sh>) is an open source container batch scheduling project that aims to simplify running multi-Pod jobs. It provides diverse scheduling policies such as:

- Gang scheduling – Schedule all or none. All pods in a job are scheduled together, preventing partial execution that could lead to resource wastage.
- Fair-share scheduling – Prevents any single job from monopolizing the cluster.
- Queue scheduling - Manages job priorities within queues. Jobs are placed in queues based on their priority, and resources are allocated accordingly to ensure high-priority jobs are executed first.
- Preemption scheduling - High-priority jobs preempt lower-priority ones. This ensures that critical jobs can interrupt less important ones to meet urgent computational needs.
- Topology-based scheduling - Considers the physical layout of nodes. This policy optimizes job placement based on node topology to improve performance and resource utilization.
- Reclaim - Reclaims resources from lower-priority queues. When new high-priority jobs enter the queue, resources can be reclaimed from less critical jobs to accommodate them.
- Backfill - Utilizes idle resources by scheduling smaller jobs. This helps maximize resource utilization by filling in gaps with smaller tasks that can run without delaying larger jobs.
- Resource reservation - Reserves resources for future jobs. This policy ensures that resources are available for scheduled jobs, preventing resource contention and ensuring smooth execution.

Volcano has two key concepts:

- PodGroup
- Queue

A PodGroup is a group of pods with a strong association. They are always scheduled together. A PodGroup specifies its minimum Pod count. If the cluster cannot meet the demand of running the minimum number of Pods, no Pod in the PodGroup gets scheduled.

A queue schedules PodGroups as they get scheduled. PodGroups can have different weights. A queue can also prioritize PodGroups with higher weightage. A queue puts a limit on the total number of resources available to your workloads and gives you a way to categorize workloads by their importance.

Conceptually Yunikorn, Kueue, and Volcano are very similar. Once you understand one of them, you'll be able to quickly adapt to the others. They all provide APIs for scheduling groups of Pods and assigning them to queues, allowing you to define desired workflows, resource allocations, and task dependencies.

After enabling gang scheduling, Kubeflow creates a PodGroup (instead of creating master and worker Pods) whenever you create a TFJob, PyTorchJob, or MPIJob. Once the PodGroup is created, Volcano creates Pods if the cluster has enough capacity to run all (master and worker) Pods simultaneously, otherwise not.

Let's see this in action. To install Volcano in your cluster, run:

```
$ helm repo add volcano-sh https://volcano-sh.github.io/helm-charts  
$ helm repo update  
$ helm install volcano volcano-sh/volcano --version 1.9.0 -n  
volcano-system --create-namespace
```

Create a queue (listing 5.6) that limits GPU usage to a maximum of 3.

```
$ cd Chapter 5/volcano  
$ kubectl apply -f queue.yaml  
queue.scheduling.volcano.sh/training-queue created
```

Listing 5.6 A Volcano queue called “training-queue” (volcano/queue.yaml)

```
apiVersion: scheduling.volcano.sh/v1beta1  
kind: Queue  
metadata:  
  name: training-queue  
spec:  
  weight: 1  
  reclaimable: false  
  capability:  
    nvidia.com/gpu: 3
```

We then configure the Kubeflow to use Volcano for gang scheduling. This is done by editing Kubeflow Training Operator's Deployment and adding “`--gang-scheduler-name=volcano`” to the controller's command line arguments.

```
$ kubectl -n kubeflow edit deployments training-operator
```

Listing 5.7 shows the snippet after adding the argument.

Listing 5.7 Training operator argument for gang scheduling

```
...  
    spec:  
      containers:  
        - command:  
          - /manager  
+          - --gang-scheduler-name=volcano #A  
        image: kubeflow/training-operator  
        name: training-operator  
...  
#A Add this line to enable gang scheduling in Kubeflow
```

With gang scheduler configured, Kubeflow automatically creates a new type of Custom Resource called PodGroup when a user creates a training job. The *PodGroup* CRD is defined and managed by Volcano, and it provides a way to manage a group of Pods as a single unit. A PodGroup acts as a logical unit that enforces the "all-or-nothing" scheduling policy for distributed training workloads.

When you submit a distributed training job (like a PyTorchJob), Kubeflow creates a PodGroup instead of worker and master Pods. This PodGroup specifies a `minMember` value equal to the job's total replica count (e.g., workers + master). For example, a TFJob requesting 3 workers and 1 parameter server would create a PodGroup with `minMember: 4`. Volcano uses this value and all the Pods in the training job are created simultaneously, or none at all if the cluster doesn't have sufficient capacity.

For instance, remember the PyTorchJob we showed in listing 5.3? If you create the same job after enabling support for gang scheduling in Kubeflow, you'll notice that a new PodGroup appears when you create the job. To view existing PodGroups, run:

```
$ kubectl get podgroups  
NAME                      STATUS     MINMEMBER  
RUNNINGS     AGE  
pytorch-dist-mnist-nccl    Running   2  
22s
```

NOTE In Amazon EKS, you may have to create a node group with GPUs in your cluster by running the `create-gpu-mng.sh` script in the book's GitHub repo.

This training job uses two GPUs. Since the queue is limited to three GPUs, if we were to create another training job that requires two more GPUs, Volcano doesn't schedule the second job until the first one finishes. The second job remains in "Pending" state until the cluster has at least two GPUs available.

```
$ kubectl get podgroups  
NAME                      STATUS     MINMEMBER  
RUNNINGS     AGE  
pytorch-dist-mnist-nccl    Running   2  
2
```

```
22s
pytorch-dist-mnist-nccl_dup          Pending   2           2
10s
```

Alternate schedulers like Volcano allow you to customize workload scheduling in Kubernetes. With a queue, we can limit cluster autoscaling and ensure that workloads are only scheduled when capacity is guaranteed. This prevents situations where training cannot initiate because the cluster doesn't have the capacity to schedule all Pods required for the training.

When your organization has a large number of jobs competing for finite resources in a Kubernetes cluster, these schedulers help you achieve more predictable and efficient workload scheduling. They improve your cluster's performance and resource management.

The integration of Kubeflow and Volcano enables teams to efficiently train and fine-tune models at scale while maximizing resource utilization. Kubeflow Trainer abstracts the complexity of distributed training by providing framework-specific custom resources (e.g., PyTorchJob, TFJob, MPIJob), while Volcano's advanced scheduling policies -- such as gang scheduling, fair-share allocation, and priority preemption -- ensure optimal cluster resource management. This cluster can handle everything from small fine-tuning jobs to large-scale distributed training across multiple nodes. Being framework agnostic, this setup provides support for multiple ML frameworks, which allows data scientists to use their preferred tools and libraries.

NOTE If you created a GPU node group for Volcano, remember to delete the node group by running: `aws eks delete-nodegroup --cluster-name mlops-cluster --nodegroup-name gpu_nodes`.

5.7 *Optimizing distributed training*

Training large models across multiple machines is both resource-intensive and costly, especially given the high expense of renting GPUs in the cloud. The ML community is continuously developing new techniques to improve the efficiency and cost-effectiveness of distributed training. Below is a list of common ways of speeding up model training.

PLACE WORKER NODES AS CLOSELY AS POSSIBLE

Slow interconnectivity between worker nodes increases the amount of time GPUs waste while moving data between worker nodes. To reduce GPU wastage, ensure that the worker nodes have low latency, high throughput connectivity. If you're in the cloud, place all worker nodes in the same Availability Zone. You'd reduce internode latency and eliminate any inter-AZ data transfer charges.

You can use Kubernetes Pod Affinity to collocate workers in an AZ by adding this configuration to a PodSpec:

```
affinity:  
  podAffinity:  
    requiredDuringSchedulingIgnoredDuringExecution:  
      - labelSelector:  
          matchLabels:  
            job: distributed-training  
        topologyKey: topology.kubernetes.io/zone
```

USE HARDWARE ACCELERATION

Another way to improve GPU utilization in distributed training is by interconnecting them with high speed network connection. Network interfaces such as NVLink and InfiniBand can significantly reduce the communication latency and increase the data transfer rate between GPUs, allowing for more efficient data exchange and synchronization during training.

NVLink is a high-speed interconnect developed by NVIDIA, which enables direct GPU-to-GPU communication at speeds of up to 100 GB/s. This allows multiple GPUs to work together seamlessly, sharing data and gradients without the need for CPU intervention. NVLink is particularly useful for large-scale distributed training, where multiple GPUs need to communicate with each other to perform complex computations.

InfiniBand, on the other hand, is a high-speed networking technology developed by Mellanox (now part of Nvidia). It provides a scalable and reliable interconnect solution for high-performance computing (HPC) applications, including distributed deep learning training. InfiniBand networks can achieve speeds of up to 200 GB/s, making them an attractive option for large-scale distributed training.

Nvidia has also developed GPUDirect Storage that allows data to be stored and accessed directly on NVIDIA GPUs, bypassing the need for CPU intervention. This drastically improves data throughput and reduces storage bottlenecks, especially when handling large datasets in distributed training environments. Several cloud storage services, like Amazon FSx for Lustre, now support GPUDirect Storage. While they may have higher upfront costs, they can potentially reduce overall training time by optimizing GPU resource utilization, potentially leading to cost savings through shorter GPU instance runtimes

TUNING NCCL TO FULLY UTILIZE CROSS-HOST NETWORKS

The Nvidia Collective Communication Library (NCCL) is crucial for efficient communication in distributed training environments, especially when using multiple GPUs, which may be attached to different nodes. One way to enhance its performance is by tuning the `NCCL_SOCKET_NTHREADS` parameter, which controls the number of threads used for socket communication. Tuning this

parameter can help optimize throughput by increasing the parallelism of network operations. The optimal number of threads varies based on the specific network and hardware configuration.

GRADIENT AGGREGATION IN FLOAT16

Using float16 precision for gradient aggregation can significantly reduce the amount of data transferred during the synchronization phase, leading to faster communication between nodes. This is particularly beneficial in distributed settings where bandwidth can be a bottleneck. By reducing the data size, float16 aggregation minimizes network congestion and accelerates the overall training process. Many GPUs natively support float16 operations for improved performance and reduced memory usage. However, care must be taken to ensure that the reduced precision does not adversely affect model accuracy, which can be managed by maintaining critical operations in higher precision.

OVERLAPPING BACKWARD PATH COMPUTATION WITH GRADIENT AGGREGATION

To further enhance the efficiency of distributed training, overlapping the backward pass computations with gradient aggregation can be employed. This technique involves initiating the gradient aggregation process while the backward computations are still ongoing. You can find an example implementation here: https://pytorch.org/tutorials/intermediate/optimizer_step_in_backward_tutorial.html. By doing so, the time spent waiting for gradient synchronization is reduced, as some of the communication happens concurrently with computation. This overlap can be achieved through careful scheduling and pipelining of tasks, allowing for better utilization of computational and network resources, thus reducing idle times and improving overall throughput.

5.8 Cleanup

Before proceeding, remember to cleanup any lingering jobs wasting your money.

Delete all jobs by running:

```
$ kubectl delete tfjob <jobname>
$ kubectl delete pytorchjob <jobname>
```

Verify that you don't have any jobs running:

```
$ kubectl get pytorchjobs.kubeflow.org,tfjobs.kubeflow.org
No resources found in default namespace.
```

Should any Pods linger, you can force delete them using the --force option. To see which Pods in the request a GPU run:

```
$ kubectl get pods -A -o custom-
columns="NAMESPACE:.metadata.namespace,POD:.metadata.name,GPUs:.
spec.containers[*].resources.requests.nvidia\com/gpu" | grep -v
"<none>"
```

Then delete the Pod (or the Kubeflow Job that created that Pod):

```
$ kubectl delete pod <podname> --force
```

Ensure that you're not running any GPU nodes by running.

```
$ kubectl get nodes -o custom-
columns="NAME:.metadata.name,GPUs:.status.allocatable.nvidia\co
m/gpu" | grep -v "<none>"
```

If you do see nodes running, find out what's running on the nodes and terminate any Pods that might be keeping it from getting terminated. To find out which Pods run on a particular node, run:

```
$ kubectl get pods -A -o custom-
columns="NAMESPACE:.metadata.namespace,POD:.metadata.name,NODE:.
spec.nodeName"
```

Summary

- Distributed training is technique to speed up model training when the model or its input data is too large to fit on a single machine. It is also useful in reducing the overall training time.
- ML libraries like PyTorch and TensorFlow provide abstraction to run multi-GPU, multi-node training.
- Data parallelism and model parallelism are two methods of parallelizing model training across multiple devices.
- In distributed training, multiple instances of training process intercommunicate synchronously using collective communications or asynchronously using a parameter server.
- Kubeflow Trainer automates the task of setting up the environment for distributed training in a Kubernetes cluster.
- MPI is often used in conjunction with Horovod as a framework agnostic tool to facilitate distributed training.
- Alternative Kubernetes schedulers like Volcano, YuniKorn, and Kueue enable strategies like gang scheduling, which reduces resource wastage by scheduling workloads only when the cluster has capacity to run a collection of resources.
- Distributed training jobs should aim to maximize GPU utilization, which can be done at software and hardware level.

6

Distributed computing with Ray and Kubernetes

This chapter covers:

- Introduction to Ray
- Setting up Ray clusters on Kubernetes
- Distributing model training with Ray Train
- Optimizing hyperparameters with Ray Tune
- Tracking experiments with MLflow
- Serving models using Ray Serve

In Chapter 5, we explored how to use Kubernetes for single/multi-node model training. We saw how Kubeflow automates setting up a cluster of model trainers in Kubernetes, streamlining the process of managing complex machine learning workflows. As far as the extent of the Kubeflow project is concerned, we've barely scratched the surface. Its capabilities extend far beyond model training. We shall return to Kubeflow, but for now, we take a detour to discuss Ray, which is a versatile framework that uses Kubernetes for distributing data analytics and machine learning workloads.

6.1 *Introduction to Ray*

Ray is a distributed computing framework that has gained significant traction in the machine learning community due to its simplicity and scalability. It offers a unified API for distributed computing that's especially useful when handling data analytics and machine learning workloads at scale.

Ray's promise is that you can develop Python code on your laptop and then scale the same code across a cluster of machines with minimal changes. This flexibility and ease of use have made Ray particularly attractive for data scientists

and machine learning engineers who want to scale their workloads without getting bogged down in the complexities of distributed systems. When used in conjunction with Kubernetes, it provides a scalable compute layer for parallelizing Python applications.

Seamless scalability is one of Ray's key strengths. It allows data scientists and machine learning engineers to focus on their algorithms and models rather than worrying about the intricacies of distributed systems. By abstracting away the complexities of distributed computing, Ray enables developers to write code that can run efficiently on a single machine or a large cluster without significant modifications.

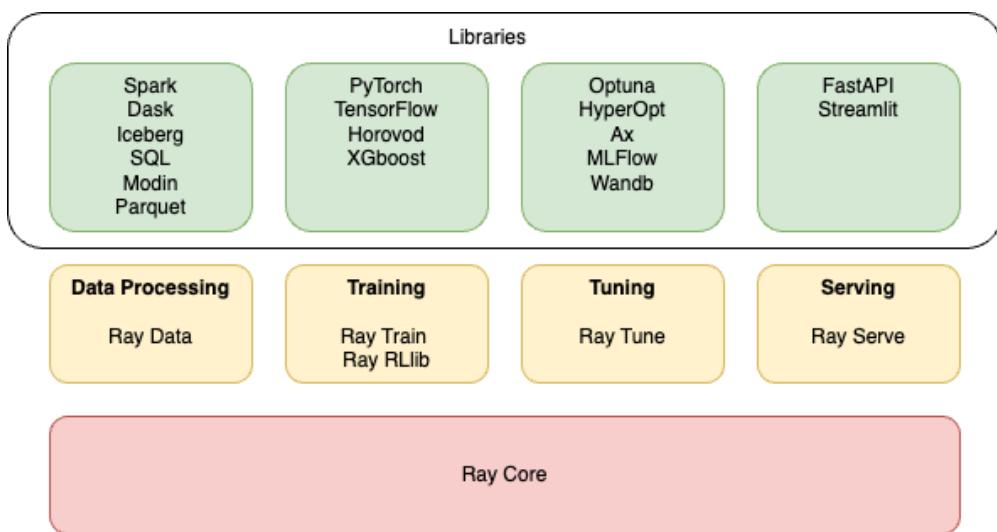


Figure 6.1 Ray's distributed computing libraries provide support for scaling a variety of data analytics and machine frameworks.

Even though Ray is a general-purpose distributed computing framework that provides a unified API, it is increasingly a popular choice for data and ML scientists. It integrates with ML libraries like PyTorch and TensorFlow to simplify distributed training. It offers libraries suited for distributed model training, reinforcement learning, hyperparameter tuning, and model serving.

Ray isn't inherently Kubernetes-native. *KubeRay*, a subproject of Ray, makes it easy to deploy and manage Ray clusters and applications on Kubernetes. On Kubernetes, a Ray cluster is made up of a group of Pods (as opposed to virtual machines in a non-containerized environment). KubeRay also integrates Ray with

other Kubernetes tools to provide support for observability, alternate scheduling, and autoscaling.

KubeRay provides Custom Resource Definitions (CRDs) to create:

- RayCluster - KubeRay fully manages the lifecycle of RayCluster, including cluster creation/deletion, autoscaling, and ensuring fault tolerance.
- RayJob – KubeRay creates a Ray cluster, runs the job, and deletes the cluster after the job finishes.
- RayService – KubeRay creates a Ray cluster that serves one or more machine learning models.

Ray is an incredibly valuable tool in the MLOps engineer's arsenal. It is especially useful for training and serving models of any size. For the rest of the chapter, we'll focus on setting up Ray clusters on Kubernetes using KubeRay, performing distributed training, and serving models. We'll apply the distributed training techniques we explored in Chapter 5 and contrast their applications when using Ray.

6.1.1 Anatomy of a Ray cluster

In earlier chapters, we've seen that Spark, PyTorch, and TensorFlow follow a model with one coordinator or scheduler that manages tasks across a group of workers. Ray also employs this model.

A Ray cluster consists of a Ray head node with one or more workers. The *head node* is identical to other worker nodes, except that it also runs singleton processes responsible for cluster management, such as autoscaling, configuration management, and the Ray driver process.

The *Ray driver* is essentially a special type of worker process that runs the top-level application script in a Ray job. It is responsible for initiating and managing the execution of a Ray application (also referred to as a job), which is a collection of tasks, actors, and objects. The individual tasks then run on worker processes on worker nodes.

While the driver orchestrates tasks and manages the application logic, worker processes are dedicated to executing the tasks assigned to them. Workers do not manage the cluster or schedule tasks; they simply perform the computations as directed by the driver.

The *Ray autoscaler*, which runs as a sidecar container with the head node, uses Kubernetes API to add and remove workers as needed.

The head node also runs the *Global Control Service* (GCS) process, which stores cluster-level metadata pertaining to task scheduling, resource management, and distributed state synchronization. By default, the GCS runs on the head node and stores data in memory. This means if the head node in a

cluster fails, the cluster loses its state. Ray supports externalizing its metadata storage into a Redis cluster, which makes a Ray cluster more fault-tolerant.

Besides the worker process, which runs tasks, all nodes run *raylets*, which is a process internal to Ray. Raylets tell the worker process on the node the tasks it should run.

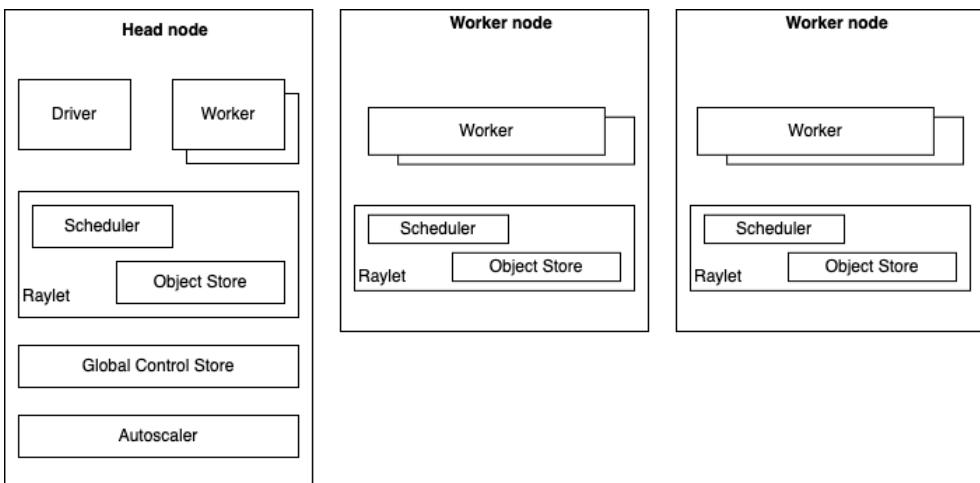


Figure 6.2 A Ray cluster with two worker nodes. Each node runs Ray helper processes to facilitate distributed scheduling and memory management. The head node runs additional control processes.

In Ray, tasks are the basic unit of computation. They are functions that are executed asynchronously and can be distributed across different nodes in the cluster. Tasks are stateless and can be executed independently. A raylet schedules tasks based on resource availability and dependencies. We use them to parallelize computations that do not require a shared state, such as data processing or batch computations.

Ray actors are stateful workers that extend the functionality of tasks by maintaining state across method calls. You create an actor by decorating a class as shown in listing 6.1. Each actor runs on a specific worker process, and its methods are executed sequentially. This allows actors to maintain internal state and handle long-lived tasks. When a new actor is instantiated, a new worker is created, and methods of the actor are scheduled on that specific worker and can access and mutate the state of that. They are suitable for scenarios that require state persistence, such as managing stateful services, model serving, or handling session data.

Listing 6.1 A rudimentary Ray actor that prints a message

```
import ray

ray.init() #A

@ray.remote #B
class Greet:
    def __init__(self):
        self.text = "hello from Ray cluster"
    def say_hello(self):
        return self.text

a = Greet.remote() #C

print(ray.get(a.say_hello.remote())) #D

# Prints "hello from Ray cluster"
#A Initialize Ray
#B The @ray.remote decorator makes Counter a Ray actor.
#C Create an actor instance
#C Create an actor
#D Call methods on the actor. ray.get() retrieves results from remote method calls.
```

Tasks and actors create and compute on objects using Ray's get and put APIs. Objects exist in Ray's distributed shared-memory object store, and every node has an object store (managed by raylets). An object can live on one or more nodes.

6.1.2 Setting up a Ray cluster with KubeRay

A Ray cluster is a place where ML and data scientists can submit tasks like data processing, model training, and hyperparameter tuning. You will create two types of Ray clusters depending on your needs:

- Persistent Ray clusters – These clusters are always running. They will be used for batch processing a dataset or developing a model.
- Ephemeral Ray clusters – These clusters are spun up to run a particular application such as batch processing of a dataset, running a model training job, or serving a model over the network. The cluster's lifecycle is tightly coupled with the application's.

Ephemeral clusters can be short or long-lived. What makes them different from persistent clusters is that they only run a single application. When the application terminates, the cluster gets torn down.

To create a Ray cluster on Kubernetes, first we must install the KubeRay operator. We can use Helm for that:

```
$ helm repo add kuberay https://ray-project.github.io/kuberay-helm/
$ helm repo update
$ kubectl create namespace ray
$ helm install kuberay-operator \
  kuberay/kuberay-operator \
  --version 1.1.1 \
  --create-namespace \
  --namespace ray
```

Deployment of Ray on Kubernetes follows the operator pattern. To create a Ray cluster, you create a `RayCluster` Custom Resource. Within which you describe the desired state of your cluster. When you create a `RayCluster` Custom Resource (CR), the KubeRay operator creates Pods based on the cluster's specifications.

To deploy a `RayCluster` CR, run:

```
$ helm install \
  raycluster \
  kuberay/ray-cluster \
  --version 1.1.1 \
  --namespace ray \
  --set 'image.tag=2.34.0-py311-cpu'
```

And with this, you've now created a Ray cluster with a head and a worker Pod. `kubectl get rayclusters -n ray` shows the current cluster configuration.

```
kubectl get rayclusters -n ray
NAME           DESIRED WORKERS   AVAILABLE WORKERS   CPUS
MEMORY
raycluster-kuberay   1             1                  2
3G
```

Let's see what we can do with this shiny new Ray cluster.

6.2 Running workloads on a Ray cluster

There are two common patterns for interacting with a Ray cluster:

- Using API – This is useful when you want to run some parts of your code remotely on a Ray cluster.
- Using ray CLI – Used when you want to run a Python script on a Ray cluster.

These two interaction patterns provide flexibility in how you leverage Ray's distributed computing capabilities. The API approach allows for fine-grained control over which parts of a program are executed remotely, enabling you to

optimize performance by selectively distributing computationally intensive tasks. The CLI method offers a straightforward way to run entire scripts on a Ray cluster, making it ideal for batch processing or long-running jobs that don't require frequent local interaction.

6.2.1 Using Ray interactively from notebooks

Let's understand how to use Ray's Python API by showing you an example. We shall return to JupyterHub and start a new Jupyter notebook by choosing the "All Spark environment" profile. A notebook with the steps in this section is also available in the book's GitHub repository.

First, we install Ray libraries using our notebook:

```
!pip install -U ray[default]==2.34.0
```

Now we need to connect the notebook's environment to the Ray cluster we deployed in the previous section. To do that, we need the IP address of the Ray cluster. Along with a Ray cluster Pods, KubeRay also creates a Kubernetes Service so operators and developers can remotely connect to a Ray cluster. Since this is a ClusterIP Service by default, it doesn't support any connections from outside the cluster. For example, we cannot connect to this Ray cluster from a notebook server running on our laptops.

To get the IP address of your Ray cluster, run from your local terminal (not in Jupyter):

```
$ kubectl -n ray get svc
NAME           TYPE      CLUSTER-IP
EXTERNAL-IP
kuberay-operator   ClusterIP  172.20.216.52
<none>
raycluster-kuberay-head-svc   ClusterIP  172.20.118.192
<none>
```

The first service is the KubeRay operator. The second service `raycluster-kuberay-head-svc` is the Ray Cluster.

We can then return to our notebook, set the Ray cluster's address (including the default client port 10001), and connect to it using:

```
ray.init(address="ray://172.20.118.192:10001")
```

Consider the code in listing 6.2. It creates a list of eight items. Then we write a function that returns each item. We've artificially slowed the function by waiting one second before returning the result. When we execute this program using our notebook, it takes eight seconds to complete. That's because our program is

limited by Python's Global Interpreter Lock (GIL) and must go through each item sequentially.

Listing 6.2 A sample program without parallelization

```
import time

database = [
    "Learning", "Ray",
    "Flexible", "Distributed", "Python", "for", "Machine",
    "Learning"
]

def print_runtime(input_data, start_time):
    print(f'Runtime: {time.time() - start_time:f} seconds,
data:')
    print(*input_data, sep="\n")

def retrieve_task(item):
    time.sleep(1)
    return item, database[item]

start = time.time()
object_references = [
    retrieve_task(item) for item in range(8)
]
data = (object_references)
print_runtime(data, start)
```

The output is:

```
Runtime: 8.000879 seconds, data:
...
```

In Listing 6.3, we've modified the code to run on the Ray cluster. Because the cluster has two available workers (head + worker), Ray runs the `retrieve_task()` function across two workers concurrently. As a result, the code finishes running in half the time.

What did we have to modify?

9. Decorate the `retrieve_task()` function with `@ray.remote`.
10. We added `.remote()` when calling the `retrieve_task()` function.

Listing 6.3 Parallelizing with Ray

```
import time
import ray

database = [
    "Learning", "Ray",
    "Flexible", "Distributed", "Python", "for", "Machine",
    "Learning"
]

def print_runtime(input_data, start_time):
    print(f'Runtime: {time.time() - start_time:f} seconds,
data:')
    print(*input_data, sep="\n")

@ray.remote
def retrieve_task(item):
    time.sleep(1)
    return item, database[item]

start = time.time()
object_references = [
    retrieve_task.remote(item) for item in range(8)
]
data = ray.get(object_references)
print_runtime(data, start)
```

When we run the code shown in listing 6.3 in our Jupyter notebook, the output is:

```
Runtime: 4.227692 seconds, data:
...
...
```

What do you think? Isn't it easy to parallelize Python applications with Ray? We're just getting started. Ray has excellent documentation, and we recommend learning more about how Ray can enhance your applications. Explaining Ray's APIs is outside the scope of this book. We'll restrict the text to teaching how you can run workloads with Ray on Kubernetes.

6.3 Customizing a cluster using RayCluster

Most organizations will need at least one Ray cluster for users that use it interactively during development or to run ad hoc jobs. Ray doesn't isolate workloads within a cluster. For this reason, we must create separate Ray clusters for workloads that require isolation.

Keep in mind that when running batch jobs (including model training/tuning) or serving models, you'll typically create ephemeral clusters using RayJob and RayService Custom Resource Definitions (CRD).

In section 6.1.2, we created a cluster, but we didn't specify things like how many workers we wanted. Neither did we specify if the cluster should have any GPUs. Let's go through the RayCluster CRD to understand how we can customize the cluster to match our needs.

Ray recommends giving a minimum of 8 CPUs and 32 GB memory to head nodes. How you size your Ray cluster Pods beyond the recommended resources will depend on your particular use case. A telltale sign of an under-provisioned Ray cluster (which can also be difficult to troubleshoot) is cluster crashes when running heavy data processing or model training tasks. To increase the size of a Ray cluster, you'll allocate more CPU units, GPUs, and memory (scale vertically) to its workers and/or add more workers (scale horizontally).

The RayCluster CRD has the following key sections:

- `rayVersion` – Specifies the cluster's version. It is also used to autofill additional config fields in the CRD.
- `headGroupSpec` – The configuration values for the head Pod. Here you can adjust values like CPU, memory, environment variables.
- `workerGroupSpecs` – The configuration values for the worker Pod(s). You can specify scaling configuration to control the number of replicas.

NOTE. The container image for head and worker Pods should match with the Ray version specified in `rayVersion`. If your workers have GPUs, you'll also need an image with CUDA runtime installed. You can browse Ray images at DockerHub (<https://hub.docker.com/r/rayproject/ray>).

To allocate CPU, memory, and GPU resources to workers or head, KubeRay uses Kubernetes resource limits and requests. Listing 6.4 contains a RayCluster spec, that creates a cluster with two replicas. We've specified the one and three as the min and max replica count respectively.

Listing 6.4 A RayCluster (persistent-raycluster.yaml)

```
apiVersion: ray.io/v1
kind: RayCluster
metadata:
  name: persistent-raycluster
  namespace: ray
spec:
  rayVersion: '2.34.0'
  enableInTreeAutoscaling: true
  autoscalerOptions:
    upscalingMode: Conservative
    idleTimeoutSeconds: 120
  headGroupSpec: #A
  rayStartParams: {}
```

```

template:
  spec:
    containers:
      - name: ray-head
        image: rayproject/ray:2.34.0-py311-cpu
        resources:
          ...
workerGroupSpecs: #B
  - groupName: workergroup #C
    maxReplicas: 3
    minReplicas: 1
    numOfHosts: 1
    rayStartParams: {} #D
    template:
      spec:
        containers:
          - image: rayproject/ray:2.34.0-py311-cpu
            name: ray-worker
            resources:
              ...
#A Head node configuration
#B Worker node configuration
#C Worker group name
#D Start parameters for Ray

```

This creates a cluster with two worker nodes. If we run many jobs in parallel, at some point the cluster's compute resources will get saturated. When this occurs, the Ray Autoscaler will add worker Pods. It will also delete workers if idle for over 120 seconds.

Setting `maxReplicas` to 3 ensures the cluster can add up to three workers. When the cluster is busy with many jobs and it cannot add any more workers, jobs will be queued.

To specify compute resources, you'll define them as Kubernetes requests and limits:

```

spec:
  workerGroupSpecs:
    - groupName: workergroup
      template:
        spec:
          containers:
            resources:
              limits:
                cpu: "8"
                memory: "32G"
                nvidia.com/gpu: 1

```

Ray prefers when you let a worker has access to the entire node. This is because it performs better when you run fewer larger workers than many smaller

workers. So, plan on having each Ray worker run on its own node. With this configuration, you'll run one Ray worker per Kubernetes node.

You can specify the number of Kubernetes nodes KubeRay should run the workers on using the `numOfHosts` field.

Our current Ray cluster is good for running small data processing jobs. It won't be able to do things like distributed model training though. We'll need a beefier cluster for those jobs. Let's tear down the current cluster and create a cluster that with more CPU and memory resources.

```
$ helm uninstall -n ray raycluster
```

To deploy a Ray cluster with recommended CPU and memory resources, run:

```
$ cd "Book/Chapter 6"  
$ kubectl apply -f persistent-raycluster.yaml
```

KubeRay will create a Ray cluster with a head and a worker. This cluster has 16 CPUs, 64 GB memory, and no GPUs.

```
$ kubectl -n ray get rayclusters.ray.io persistent-raycluster -o  
jsonpath-as-json=".status"  
  
{  
    "availableWorkerReplicas": 2,  
    "desiredCPU": "16",  
    "desiredGPU": "0",  
    "desiredMemory": "64G",  
    "desiredWorkerReplicas": 1,  
    ...  
}
```

KubeRay also creates a Kubernetes Service to provide access to the Ray cluster dashboard service running in the head Pod. This Service is typically not exposed outside the cluster although you can create an Ingress to make it accessible from outside the cluster. You can use kubectl to access the Service without attaching a load balancer.

To access the Ray dashboard, run this command from your machine (and not from the Jupyter notebook) to setup a port forwarding to the Ray head service:

```
$ kubectl -n ray port-forward \  
  svc/persistent-raycluster-head-svc 8265:8265
```

Then open "http://localhost:8265" in your browser to open the Ray dashboard.

Figure 6.3 Ray dashboard showing the cluster. The dashboard is used for introspection and debugging primarily.

NOTE. Ray’s documentation recommends restricting access to the cluster endpoint, including the dashboard. That’s because Ray allows any clients to run arbitrary code. We strongly advise limiting access to Ray using network isolation methods (such as Kubernetes network policies or security groups in AWS).

6.4 Running jobs in a Ray Cluster

You’ve seen how to connect to a remote Ray cluster using API in a notebook. Now let’s see how to run a Python script on an existing cluster using the “ray” command line tool.

Assume you have a vanilla PyTorch model training script, and you want to run it on your Ray cluster. You can run the script on an existing Ray cluster using the Ray Job CLI API.

```
$ ray job submit --working-dir=<parent_folder>/ -- python
<script_name>.py
```

The `working-dir` is the name of the directory that contains your Python file and any dependencies. The Ray CLI client zips this directory and transfers it to the Ray cluster. This ensures that any file dependencies are included wherever this script is run. The `working-dir` can be a local directory or a remote URI to a .zip file (support includes Amazon S3, Google Cloud Storage, and HTTP).

If your script requires a Python library that's not installed in the Ray container image, you can specify it in your script's runtime environment. In Ray, the runtime environment contains any additional packages your application needs. It is also a place to define any environment variables needed to run the code. The things you can customize using the runtime environment include:

- `working_dir` – Directory to be uploaded to all workers. It can be a local directory, a zip archive, or a URI to a remotely stored zipped archive. Ray Jobs API supports providing `working_dir` as a standalone parameter or part of the runtime environment configuration.
- `pip` or `conda` – any packages to be installed on the fly using pip or conda.
- `env_vars` – Environment variables to set.

With runtime environment configuration, you define how Ray should run your code. You can also customize the environment when interacting with Ray using a notebook. The configuration supports YAML and JSON.

So if your script needs TensorFlow, you can tell Ray to install it at runtime:

```
$ ray job submit \  
  --runtime-env-json='{"working_dir": ".", "pip": \  
    ["tensorflow==2.17.0"]}' \  
  -- python my-tensorflow-script.py
```

NOTE. In production, we recommend creating a custom image for Ray with application dependencies baked in. This reduces the job startup time as Ray doesn't have to install dependencies on the fly.

This is a good way to run any Python script that you'd rather not run locally. But this script isn't set up to take advantage of Ray's parallelization capabilities yet.

To take advantage of Ray's distributed computing API, we must make use of Ray libraries. The changes you must make to a particular script depend on the task at hand. For example, to parallelize a training job, you'd use the Ray Train library. Similarly, Ray Data helps you process large amounts of data in parallel.

Consider the model training script "pytorch-sample.py" in the book's GitHub repo. This script trains a model using PyTorch DistributedDataParallel (<https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html>) to run a distributed training job across ten workers. It includes changes required to have Ray run the training job across multiple nodes.

First, get the IP of the Ray head service, which gets created during Ray cluster creation:

```
$ kubectl -n ray get svc persistent-raycluster-head-svc \
    -o jsonpath='{.spec.clusterIP}'
172.20.240.111
```

In your Jupyter notebook, open a new terminal session and configure the Ray cluster address:

```
$ export RAY_ADDRESS="http://<Your ray-head Service IP or
name>:8265"
```

Download the training script from the book's GitHub repository:

```
$ curl "https://raw.githubusercontent.com/mlops-on-
kubernetes/Book/refs/heads/main/Chapter%206/pytorch-sample.py" -o
pytorch-sample.py
```

To kick off the PyTorch training, run:

```
$ ray job submit \
    --runtime-env-json='{"working_dir": ".", "pip": [
        "torch==2.4.0", "torchvision==0.19.0"]}' \
    -- python pytorch-sample.py
```

In `pytorch-sample.py` we've set `num_workers` to 10. Each worker process runs the training function. The `num_workers` parameter controls parallelization of the PyTorch training job. Let's understand how it interacts with our Ray cluster configuration.

Our Ray cluster consists of one head Pod and one worker Pod, each allocated 8 CPUs. This gives us a total of 16 CPUs across the Ray cluster. When Ray initializes, it creates one Ray process per CPU. Each Ray process executes one task at a time, allowing our cluster to run up to 16 tasks concurrently. When we launch our training job, Ray will distribute the workload across available processes. The 10 tasks will occupy 10 of the 16 available CPU resources, leaving 6 for other computational tasks.

If we set `num_workers` to 30, our Ray cluster won't have enough capacity to run the job initially. To meet the demand, Ray will automatically scale up, adding workers, up to a maximum of 3 (`maxReplicas=3`). Since each worker provides 8 CPUs, scaling up to 3 workers increases the cluster's total CPU count to 32 (8 from the head node + 3 × 8 from workers). This expansion ensures the cluster has enough resources to accommodate all 30 worker processes.

6.4.1 Submitting Ray Jobs using API

Besides using the Ray CLI, you can also run this program using `ray.job_submission.JobSubmissionClient` API. Listing 6.5 shows a Python script that executes the same program as the one we submitted using Ray CLI.

When you submit this job, the driver script runs on the head node. It uses Ray's job submission API to kick off a Python script on the workers.

Listing 6.5 A Ray driver script

```
from ray.job_submission import JobSubmissionClient
from os import environ

client = JobSubmissionClient()

run_torch = (
    "python pytorch-sample.py"
)

submission_id = client.submit_job(
    entrypoint = run_torch, #A
    runtime_env = {
        "pip": ["torch==2.4.0", "torchvision==0.19.0"], #B
        "working_dir": "./",
    }
)

print("Use the following command to follow this Job's logs:")
print(f"ray job logs '{submission_id}' --follow")
#A The entrypoint of the script
#B Additional libraries to be installed
```

You can start this job by running the driver script:

```
$ python ray-driver-script.py
Use the following command to follow this Job's logs:
ray job logs 'raysubmit_Vr69K2g3ebjfeQ5i' -follow
```

The script prints the job ID, which you can use to view the standard output of the script:

```
$ ray job logs 'raysubmit_Vr69K2g3ebjfeQ5i' -follow
```

Alternatively, you can view logs and track the progress of a job using the Ray dashboard.

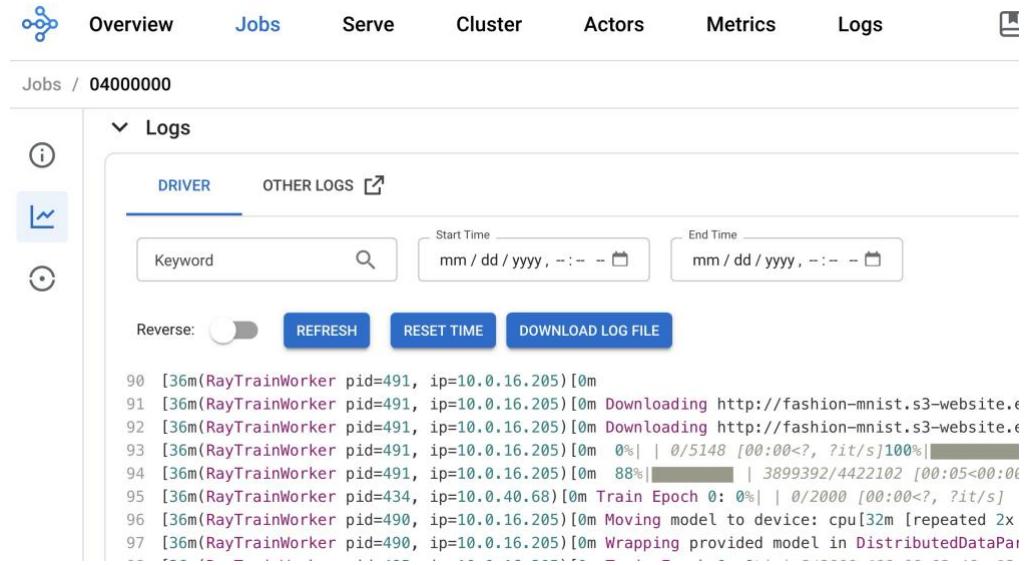


Figure 6.4 Screenshot of Ray dashboard showing logs produced by a model training job.

6.5 Adding fault tolerance to a Ray cluster

A Ray cluster persists its state and metadata in Global Control Service (GCS). By default, the GCS is not fault tolerant. It stores data in memory and if it crashes or the head Pod crashes, the cluster loses its metadata, leading to a cluster failure.

We can improve a cluster's fault tolerance by providing it with a highly available Redis cluster. This can be done by deploying a Redis cluster in the Kubernetes cluster or using a managed service like Amazon ElastiCache. Ray documentation includes a helpful guide to deploying a Ray cluster with an external Redis setup (<https://docs.ray.io/en/latest/cluster/kubernetes/user-guides/kuberay-gcs-ft.html>).

To enable fault tolerance, you annotate the RayCluster CRD with `ray.io/ft-enabled: "true"` as shown below:

```

kind: RayCluster
metadata:
annotations:
  ray.io/ft-enabled: "true"

```

When fault tolerance is enabled, should the GCS crash or restart, it loads data from Redis upon startup. Once the cluster recovers from failure, all functions are resumed. During recovery, Ray applications remain alive.

When the GCS fails, the raylet attempts reconnecting. If the raylet fails to reconnect to the GCS for more than 60 seconds (the value is configurable), the raylet will exit and the corresponding node fails.

Fault tolerance is highly recommended with using Ray to serve models, which we'll explore later in the chapter.

6.6 *Running batch workloads with RayJob*

So far, we have executed jobs on an existing Ray cluster. Imagine a team of data scientists sharing this cluster. If they all submit jobs at the same time, at some point the cluster will run out of capacity. When this happens, Ray queues jobs it cannot run (due to insufficient capacity).

Instead of having a persistent cluster that teams share, you can use the RayJob CRD to create a fresh Ray cluster for every job. *RayJob* simplifies cluster lifecycle management by handling the creation and teardown of clusters automatically. It automatically creates a RayCluster when a job is submitted and can delete it once the job is completed.

Since Ray doesn't isolate applications within a cluster, RayJob provides a mechanism to isolate jobs. Each RayJob can be configured independently, allowing for different Ray versions and configurations for each job submission. This flexibility is particularly useful for testing different setups or upgrading Ray versions without downtime.

RayJob also simplifies job submission. To trigger a job, all you need is a RayJob Custom Resource. Behind the scenes, KubeRay creates a cluster, submits the job, and tears down the cluster when the job finishes.

Listing 6.6 shows a snippet of a sample RayJob. It contains the following key elements:

- `entrypoint` – the command workers to be executed on the workers.
- `runtimeEnvYaml` – The runtime environment configuration in YAML.
- `rayClusterSpec` – The configuration of the Ray cluster including head and worker spec.

Listing 6.6 A sample RayJob (sample-rayjob.yaml)

```
apiVersion: ray.io/v1
kind: RayJob
metadata:
  name: rayjob-sample
spec:
  entrypoint: python /home/ray/code/pytorch-sample.py
```

```
runtimeEnvYAML: |
  pip:
    - torch==2.4.0
    - torchvision==0.19.0
  env_vars: {}
rayClusterSpec:
  ...
```

One key difference when running a job with KubeRay is that you cannot use Ray to package the working directory and your code. A common practice is to zip the code directory and its dependencies and place it in an object storage service that Ray supports. For example, on AWS, you can place the zip on an Amazon S3 bucket and reference it in your RayJob spec like this:

```
spec:
  runtimeEnvYAML: |
    working_dir: "s3://my_python_package.zip"
```

If your code is limited to a single Python script, you can also put the Python script in a ConfigMap, which you then mount in a RayJob Pod. For example, to put the model training script we used in section 6.2.3, run:

```
$ kubectl -n ray create configmap ray-job-code --from-file=pytorch-sample.py
```

We can then mount this Config Map and pytorch-sample.py is placed on the mount path as shown in listing 6.7.

Listing 6.7 A RayJob with a mounted script (sample-rayjob.yaml)

```
...
  volumeMounts:
    - mountPath: /home/ray/code
      name: training-script
  volumes:
    - name: training-script
      configMap:
        name: ray-job-code
        items:
          - key: pytorch-sample.py
            path: pytorch-sample.py
...

```

You can find a complete RayJob YAML in the book's GitHub repo at "Book/Chapter 6/ sample-rayjob.yaml".

To schedule this job, we just create the Custom Resource:

```
$ kubectl apply -f sample-rayjob.yaml
```

Once we create this job, KubeRay creates a Ray cluster based on the resource spec. The cluster has a head node and a worker. Once the cluster is functional, KubeRay creates a Kubernetes Job to submit the job to the Ray cluster. It does what we did manually in section 6.2.3 when we used Ray API or CLI to submit the job. If you create this RayJob, you'll see a Kubernetes Job in the ray namespace:

```
$ kubectl -n ray get pods -w
NAME                               READY   STATUS    AGE
kuberay-operator-7d7998bcdb-qjj9g   1/1     Running  10m
ob-crd-sample-raycluster-z28hf-worker-workergroup-tmx9k   1/1     Running  10m
rayjob-crd-sample-raycluster-z28hf-head-q5vz2      2/2     Running  10m
rayjob-crd-sample-t2rkd            0/1     Completed  10m
```

Once the job (Ray job, not the submitter job) completes, the cluster automatically gets torn down after 60 minutes. To turn off automatic cluster teardown, set `shutdownAfterJobFinishes` to `False`. You can also control the duration using `ttlSecondsAfterFinished`.

In our experience, the process of executing a job via RayJob can be noticeably slower compared to utilizing an existing cluster. Typically, there is a latency of about 60 to 90 seconds before the job commences, primarily due to the time-intensive tasks of creating and configuring the Ray cluster. This delay renders persistent clusters particularly advantageous for short-lived tasks. For jobs with short runtime, the repeated overhead of creating a new Ray cluster for each task can lead to significant costs. It is beneficial to have a persistent cluster (ideally one that autoscales) for interactive development and ad-hoc or short-lived jobs. When you need workload isolation or flexibility with Ray cluster configuration, choose a RayJob.

Distributed model training, fine tuning, and large data pipelines are ideal candidates for Ray Job. You can start the development in a Ray cluster, then transition it to a distributed job once the code is ready to process larger data.

6.7 Hyperparameter tuning with Ray

Besides distributed training, Ray is also a great tool for fine tuning models. *Ray Tune* offers a Python library that is especially useful for optimizing

hyperparameters. Hyperparameters are external configuration variables like learning rate, epochs, and batch size that a data scientist sets when training a model. Unlike internal parameters, which are derived during the learning process, hyperparameters are set manually.

When training a model, a data scientist may start with estimated values for hyperparameters. These initial values are typically based on experience, domain knowledge, or sometimes even educated guesses. For instance, a data scientist might start by setting the number of epochs to ten as a baseline. However, this initial value may not be optimal for the specific problem at hand. To find the ideal value, you change the hyperparameter until you find the optimal value.

Hyperparameter optimization is the process of finding the values of hyperparameters to improve a model's efficiency and accuracy. Searching for the optimal value is done by training the same model multiple times, with different hyperparameters. For example, if I want to find out if my model performs better with 20, 30, or 40 epochs, I'll train model three times each time varying the epoch and learn the value that performs better.

This process may sound straightforward in theory, but in practice, hyperparameter optimization can become complex and computationally expensive. The complexity arises because data scientists are rarely tweaking a single hyperparameter in isolation. Instead, they often need to optimize multiple hyperparameters simultaneously, each of which can significantly impact the model's performance.

For instance, in addition to the number of epochs, a neural network might require tuning of the learning rate, batch size, number of hidden layers, neurons per layer, dropout rate, and activation functions. The interactions between these hyperparameters can be intricate and non-linear, making it challenging to predict how changes in one parameter might affect the overall model performance.

Ray Tune helps you automate hyperparameter optimization and gives you API to conduct trials in parallel. It lets you define a parameter search space using an intuitive configuration system. For example, you can specify a range of learning rates using a uniform distribution, choose from a set of batch sizes, and define the number of epochs as discrete options. This flexibility allows you to explore a wide range of hyperparameter combinations efficiently.

One of the standout features of Ray Tune is its ability to conduct trials in parallel, enabling more efficient exploration of the hyperparameter space. This parallelization significantly reduces the time taken to identify optimal configurations, particularly when working with computationally intensive models. By leveraging distributed computing resources, Ray Tune can evaluate multiple hyperparameter configurations simultaneously, dramatically accelerating the optimization process.

6.7.1 Tracking experiments with MLflow

Ray Tune helps you find the best hyperparameters for a model by running multiple trials. Ray runs each trial with a discrete combination of hyperparameters. It has built-in support for many search algorithms ranging from simple random search and grid search to more sophisticated methods like Bayesian optimization, Hyperband (<https://arxiv.org/abs/1603.06560>), and Optuna(<https://github.com/optuna/optuna>).

It also integrates with MLflow, which is an open source tool for experiment tracking (and more). When used together, Ray reports a model's metrics and hyperparameters to MLflow. Data scientists can then easily visualize and compare the results of different trials through MLflow's user-friendly interface. This integration provides a comprehensive solution for managing the entire hyperparameter optimization workflow, from experiment tracking to result analysis.

The combination of Ray Tune and MLflow offers several advantages for machine learning practitioners. Firstly, it allows for detailed logging of each trial's performance metrics, hyperparameters, and even model artifacts. This comprehensive tracking is crucial for reproducibility and enables data scientists to revisit and analyze past experiments with ease.

Moreover, MLflow's web-based UI provides a user-friendly platform for exploring the results of hyperparameter optimization runs. You can quickly identify the best-performing configurations, compare different trials side by side, and visualize the relationships between hyperparameters and model performance. This visual exploration can offer valuable insights into the model's behavior and sensitivity to different hyperparameters.

The integration also facilitates collaboration among team members. By centralizing experiment tracking and results, multiple researchers can share findings, compare approaches, and build upon each other's work more effectively. This is particularly beneficial in larger organizations or research groups where multiple people may be working on similar problems.

Another significant advantage is the ability to version control your machine learning experiments. MLflow allows you to track different versions of your models along with their corresponding hyperparameters and performance metrics. This versioning capability is invaluable for maintaining a clear history of your model development process and for reverting to previous configurations if needed.

Kubernetes is a great place to run MLflow. You can quickly install MLflow in your Kubernetes cluster using Helm:

```
$ helm repo add community-charts https://community-
```

```
charts.github.io/helm-charts
$ helm repo update
$ helm install mlflow \
  community-charts/mlflow \
  --version 0.7.19 \
  --namespace mlflow \
  --create-namespace
```

Note that this deployment isn't ideal for production. MLflow requires an external Postgres or MySQL database for data persistence. For now, we'll skip creating a database for MLflow. The implication of not having a database is that if MLflow Pod crashes or restarts, its data will be lost. That's okay for this simple demonstration. In production, be sure to use a highly available database cluster so MLflow can persist its data and survive restarts. If you are in the cloud, consider using your cloud provider's managed database service (like Amazon Relational Database Service (RDS)) for MLflow database as explained in MLflow docs <https://mlflow.org/docs/latest/tracking/tutorials/remote-server/>.

To access MLflow web UI, setup port forwarding and open the UI in your web browser by going to `http://localhost:5000`:

```
$ kubectl -n mlflow port-forward svc/mlflow 5000:5000
```

This book's GitHub repo includes a Python (Chapter 6/mlflow-tune.py) script that shows an implementation of Ray Tune for hyperparameter optimization. It defines a parameter search space and creates ten trials (the number of experiments is controlled using `num_samples`) by randomly selecting hyperparameter values from the search space. Ray reports each trial's data to MLflow.

Before running this script, you'll need the IP address of your MLflow tracking server. You can get it by running:

```
$ kubectl -n mlflow get svc mlflow -o
jsonpath='{.spec.clusterIP}'
172.20.52.135
```

Since this service is of type ClusterIP, it is only accessible within the cluster. We cannot connect to it directly from my local machine, but it's accessible from our Jupyter Notebook server.

To run the script, we log into JupyterHub and download the Python script from GitHub. Place the Python file in a new folder so when Ray zips up the current directory it doesn't include unneeded files. Open a terminal in your notebook server and run:

```
$ mkdir newdir; cd newdir
```

```

$ curl "https://raw.githubusercontent.com/mllops-on-
kubernetes/Book/refs/heads/main/Chapter%206/mlflow-tune.py" -o
mlflow-tune.py
$ export RAY_ADDRESS="http://<your ray svc IP>:8265"
$ ray job submit --runtime-env-json='{"working_dir": ".", "pip": [
["mlflow==2.16.0", "torch==2.3.1", "pytorch_lightning==1.9.0",
"torchvision"]}]' -- python mlflow-tune.py --tracking-uri
http://<your mlflow service IP>:5000

```

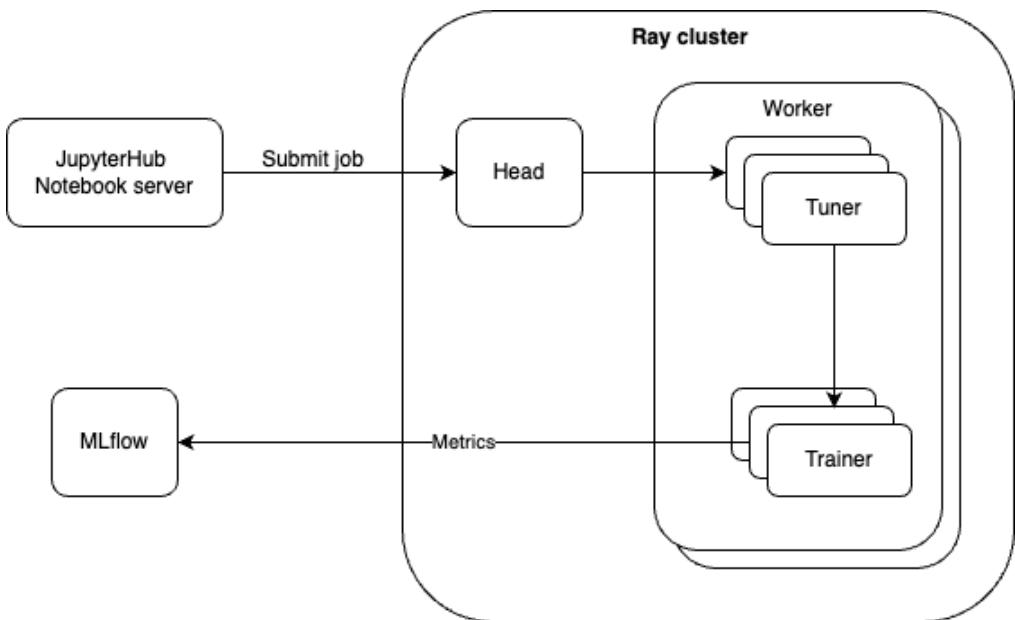


Figure 6.5 Ray Tune trains a model with different combinations of hyperparameters. The metrics of each trial are logged into MLflow for further analysis.

The job will run the script in our existing Ray cluster, which creates ten consecutive trial runs. Once the trials are complete, the best performing hyperparameters are identified. Since we did not explicitly specify a search algorithm in the script, Ray defaults to using random search and grid search, balancing exploration and exhaustive evaluation of the parameter space. When the tuning process concludes, Ray automatically logs each trial's performance metrics in MLflow, allowing us to analyze and compare results, visualize trends, and retrieve the optimal configuration for deployment.

NOTE. In the example, we submitted the job to an existing Ray cluster, which shares resources with other workloads. However, for better resource isolation and job management, we could run the script on a standalone Ray server by creating a RayJob CR. This approach allows Kubernetes to manage the job lifecycle independently from the existing cluster, ensuring dedicated resources, easier tracking, and cleaner teardown after execution. By defining a RayJob CR, we can specify job configurations, dependencies, and compute requirements, enabling more predictable performance and reduced contention with other workloads.

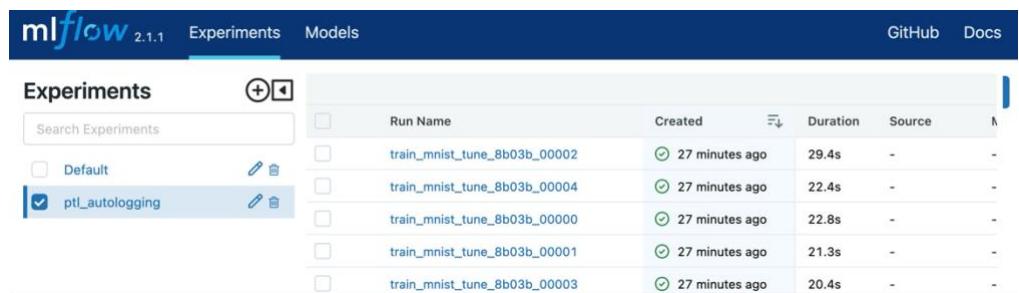


Figure 6.6 The MLflow Tracking dashboard provides a comprehensive view of the trial runs, offering insights into metrics, parameters, and model behavior.

This setup combines the distributed computing power of Ray, the experiment tracking capabilities of MLflow, and the scalability and flexibility of Kubernetes.

Experiment tracking is a critical functionality for anyone building and optimizing machine learning models. It ensures reproducibility, facilitates collaboration, and provides a historical record of model development. If you've deployed MLflow in your cluster, take a moment to explore it.

You can tear down the Ray cluster by running:

```
$ kubectl delete -f persistent-raycluster.yaml
```

6.8 Inference with Ray Serve

Another area where Ray shines is serving models. Being framework agnostic, Ray can serve models authored using a variety of ML frameworks, such as PyTorch, TensorFlow, and Scikit-Learn. *Ray Serve* is a Python library designed to handle complex serving scenarios, including multi-model pipelines, business logic integration, and high-performance inference.

KubeRay provides RayService Custom Resource to simplify model deployments. It supports high availability, blue/green deployments, monitoring,

health checking, failure recovery, autoscaling, and multi-node/multi-GPU inference.

One of the key advantages of Ray Serve is its ability to compose a pipeline that consists of multiple models. For example, you can build a Ray application that uses one model to translate text from English to French and another model to summarize the translated text. This flexibility allows for the creation of complex, multi-step inference pipelines that can handle sophisticated ML workflows.

Ray Serve also offers features like dynamic batching and response streaming, which are particularly useful for optimizing performance and handling large-scale inference tasks like Large Language Models. Dynamic batching allows the system to automatically group incoming requests for more efficient processing, while response streaming enables the server to start sending results back to the client as soon as they're available, rather than waiting for the entire computation to complete.

Like RayJob, RayService creates a Ray cluster that is used to run the Ray application. In this case, the cluster will be dedicated to serving the deployed models. The RayService manages the lifecycle of this cluster, ensuring that the necessary resources are available for model inference and that the serving infrastructure can scale as needed.

To deploy a model, all you must do is create a RayService Custom Resource. Within the RayService Custom Resource, you specify:

- Ray Serve configuration, including:
 - The application name
 - The import path for your Ray Serve deployment code
 - Any runtime environment requirements (e.g., pip packages)
- Ray cluster configuration, including:
 - Ray version
 - Head node specifications
 - Worker node specifications
 - Resource requirements (CPU, GPU, memory)
- Scaling options, such as minimum and maximum replicas

In this book's GitHub repository, you'll find `text_ml.py`, which uses Ray Serve to create an application that translates incoming text from English to French and then summarizes. In the script, we create two deployments: one for translation and another for summarization.

In Ray Serve, *deployments* represent units of business logic or ML models that handle incoming requests. At its core, a deployment consists of multiple replicas, which are individual instances of a Python class or function running in separate Ray Actors. This design allows for flexible scaling, where the number of replicas can be manually adjusted or automatically scaled to match incoming request loads, ensuring efficient resource utilization and responsive performance.

To create a deployment, you use the `@serve.deployment` decorator on a Python class or function, encapsulating the desired logic. A Serve application consists of one or more deployments. There's always one deployment, called ingress deployment, that acts as the entry point for all incoming requests. In our example, the "Translator" deployment is the ingress deployment as all requests first get translated. The results of the Translator deployment are then sent to the "Summarizer" deployment. Please see "Chapter 6/rayservice/text_ml.py" for the complete Python code.

```
@serve.deployment
class Translator:
    # Translate text
    ...

@serve.deployment
class Summarizer:
    # Summarize text
    ...
app = Summarizer.bind(Translator.bind())
```

Listing 6.8 contains a snippet of a RayService Custom Resource, which deploys the Translator-Summarizer application. The full deployment YAML file is available in the book's GitHub repository at "Chapter 6/rayservice/ray-service.text-ml.yaml".

Upon creation of this RayService, KubeRay creates a new Ray cluster and runs the specified Python code. As configured in Ray's runtime environment, Ray downloads the code from GitHub and executes `text_ml.py` module. This Python code downloads the model from Hugging Face Hub (using the Hugging Face Transformers library) and translates text from English to French. Once deployed, the RayService creates an HTTP endpoint where you can send requests. We can then send HTTP requests to the endpoint, and it will first translate text from English to French and then summarize it for us.

Listing 6.8 Snippet of a RayService Custom Resource (ray-service.text-ml.yaml)

```
apiVersion: ray.io/v1
kind: RayService
metadata:
  name: rs-text-sum
spec:
  serveConfigV2: |
    applications:
      - name: text_ml_app
        route_prefix: /summarize_translate #A
        runtime_env:
          working_dir: "https://github.com/mlops-on-
kubernetes/Book/raw/main/Chapter%206/serve-config.zip" #B
        pip: #C
          - torch
          - transformers
        import_path: text_ml.app #D
    deployments:
      - name: Translator #E
        num_replicas: 1 #F
        ray_actor_options:
          num_cpus: 0.2 #G
        user_config:
          language: french
      - name: Summarizer
        num_replicas: 1
        ray_actor_options:
          num_cpus: 0.2
  rayClusterConfig:
    headGroupSpec: {}
    workerGroupSpecs: {}

#A The base URL path for all endpoints in this application (For example,
#www.example.com/summarize_translate)
#B Defines the working directory for the application. In this case, it's set to a URL
#pointing to a zip file on GitHub. Ray will download and extract this zip file, making its
#contents available to the application.
#C Additional libraries to install at runtime
#D The name of the entry point Python file suffixed with .app
#E The name of the first deployment (used for translating text)
#F Number of translator replicas, typically each replica maps to a CPU or GPU
#G CPU resources for the translator deployment
```

To deploy this Ray application, run:

```
$ cd rayservice
$ kubectl apply -f ray-service.text-ml.yaml
rayservice.ray.io/rs-text-sum created
```

It takes 3-5 minutes for the deployment to complete. You'll know when the deployment is complete when there's a Kubernetes Service called `rs-text-sum-serve-svc` in the default namespace.

Like RayJob, when you create a RayService, KubeRay creates a vanilla Ray cluster. Once the cluster is operational, KubeRay deploys the Serve application. You can see the progress using the Ray user interface. Setup a port forwarding and then open `http://localhost:8265` to access Ray.

```
$ kubectl get services
NAME                           TYPE      CLUSTER-IP
EXTERNAL-IP    PORT(S)
kubernetes      <none>   ClusterIP   172.20.0.1
<none>          443/TCP
rs-text-sum-raycluster-6pmk8-head-svc  ClusterIP
172.20.215.233  <none>

$ kubectl port-forward svc/rs-text-sum-raycluster-6pmk8-head-svc
8265
Forwarding from 127.0.0.1:8265 -> 8265
```

Besides the Kubernetes Services KubeRay creates for the head node, it also creates a Service for the Serve application. This Service routes API traffic to the Serve application Pods. Usually, you'd add a Kubernetes Ingress or a load balancer to expose the Serve Service outside the Kubernetes cluster. For now, we can use port forwarding to test this application. To invoke the service, find out the name of the Kubernetes Service KubeRay creates (it defaults to `rs-text-sum-serve-svc`) and then setup a port forward so we can send requests to the RayService using curl:

```
$ kubectl port-forward svc/rs-text-sum-serve-svc 8000
```

Open another terminal, and send a request to the service using curl:

```
$ curl -X POST -H \
  "Content-Type: application/json" \
  localhost:8000/summarize_translate -d \
  '"Once upon a bye, there was a mischievous boy named Tom
Sawyer, who was always getting into trouble, getting off on
danger."'
> Un garçon malveillant, Tom Sawyer, s'est lancé en danger et
s'est toujours heurté à des difficultés.
```

There you have it. We have deployed an application that exposes API to translate and summarize text using pretrained machine learning models.

Ray simplifies dynamic application configuration updates without requiring a restart. For instance, if we want the application to translate into German instead

of French, we can simply update `user_config.language` to "german" and apply the changes to the RayService. Ray then automatically propagates the update across all running and future deployment replicas, ensuring a seamless transition. This capability is particularly valuable in production environments, where deploying updates without downtime is essential for maintaining availability and a smooth user experience.

Because it takes several minutes to deploy a RayService, you'd want to avoid any situations that cause a restart in production. The following parameters are safe as changing them does not cause a restart:

- `num_replicas`
- `autoscaling_config`
- `user_config`
- `max_ongoing_requests`
- `graceful_shutdown_timeout_s`
- `graceful_shutdown_wait_loop_s`
- `health_check_period_s`
- `health_check_timeout_s`

Some parameters, such as `import_path` and `runtime_env` do require a restart. If you're targeting zero-downtime updates, be sure to know which parameters can be updated dynamically (<https://docs.ray.io/en/latest/serve/advanced-guides/inplace-updates.html>).

You can tear down the Ray service along with all the other components using:

```
$ kubectl delete -f ray-service.text-m1.yaml
```

6.9 Scaling Ray Serve

When you send a request to a Ray Serve application, the request is first handled by the Serve proxy. This proxy acts as the entry point for all incoming traffic to the Ray cluster. By default, the proxy runs on every Ray node that has at least one replica actor. From there, the request is forwarded to one of the deployment replicas based on Ray's internal routing logic. When a deployment has multiple replicas, Ray routes to the replica that has fewer in-flight requests.

Ray Serve autoscaling automatically adjusts the number of replicas of a deployment based on incoming traffic. This ensures that your application can

handle varying loads efficiently, scaling up during traffic spikes and scaling down during quieter periods.

The autoscaling mechanism in Ray Serve operates by monitoring the number of ongoing requests and queue sizes. It makes scaling decisions to add or remove replicas based on these metrics, aiming to maintain a target level of performance while optimizing resource usage.

By default, Ray creates as many replicas as you define in the `num_replicas` parameter. To enable autoscaling, you can set the `num_replicas` for a deployment to "auto". Additionally, you can also specify min and max replicas to control a deployment's autoscaling behavior.

Ray autoscales a deployment based on its number of in-flight requests. For instance, suppose I want every deployment replica to handle an average of 5 concurrent requests. I can do this by setting the deployment's `target_ongoing_requests` to 5. If this deployment receives 10 concurrent requests, Ray will adjust the number of deployment replicas to 2.

Together Ray Serve on Kubernetes offers a robust and scalable solution for serving models. Ray scales the application seamlessly based on the incoming traffic by adding or removing replicas. The Ray cluster itself scales as the number of deployment replicas scales. Furthermore, Karpenter scales the Kubernetes cluster whenever the Ray cluster needs more capacity or scales down. The setup not only simplifies the deployment and scaling of ML models but also allows data scientists and ML engineers to focus on model development and improvement rather than infrastructure management.

6.10 Comparing Ray with Kubeflow

In Chapter 5, we discussed Kubeflow and the capabilities it brings to data analytics and model training tasks. Both Ray and Kubeflow provide similar functionality at the fundamental layer, offering tools for distributed computing and machine learning workflows on Kubernetes. However, they have distinct strengths and use cases that make them suitable for different scenarios.

Ray excels in providing a flexible and scalable distributed computing framework with a focus on Python-based workloads. It offers a unified API that simplifies the development of distributed applications, making it particularly attractive for data scientists and machine learning engineers who want to scale their existing Python code with minimal changes. Ray's ecosystem includes specialized libraries like RLlib for reinforcement learning, Ray Tune for hyperparameter tuning, and Ray Serve for model serving, which can be powerful tools for specific AI and ML tasks.

On the other hand, Kubeflow provides a more comprehensive MLOps platform with a broader focus on the entire machine learning lifecycle. It offers multi-user

isolation, workflow orchestration with Kubeflow Pipelines, and out-of-the-box integration with major cloud providers. Kubeflow is particularly well-suited for organizations that need a full-fledged ML platform with built-in support for collaboration, experiment tracking, and model deployment.

While Ray and Kubeflow have overlapping functionalities, they can be used together to create a more powerful and versatile MLOps environment. The benefit of Kubeflow is that you can use it as an end-to-end MLOps platform or for specific tasks like model training, serving, or data analytics (using a Spark cluster). If Kubeflow is available and supported in your environment, it will become an attractive option for building an MLOps platform. However, if Kubeflow doesn't support your cloud/hybrid environment, or if you want more control over the individual components that make up your MLOps platform, Ray is an excellent choice.

Table 6.1 A comparison of Ray and Kubeflow

Feature	Ray	Kubeflow
Primary focus	Python-based workloads, scalable distributed computing	Entire machine learning lifecycle
Key Strengths	Unified API for distributed applications Flexible architecture High performance due to in-memory computation Specialized libraries (RLlib, Ray Tune, Ray Serve) Minimal code changes for scaling	End-to-end machine learning tooling Workflow orchestration with Kubeflow Pipelines Supports complex workflows and reproducibility Built-in collaboration features
Supported Workloads	Data analytics Model training Reinforcement learning Hyperparameter tuning Model serving	Notebook hosting Data analytics Model training Model serving Experiment tracking ML and data pipelines
Cloud	Integrates with Kubernetes	Out-of-the-box integration with

integration		major cloud providers (Support may vary based on the cloud provider)
-------------	--	--

Summary

- Ray is a distributed computing framework for parallelizing Python applications.
- It offers integrations with machine learning libraries for distributed training, data processing, model tuning, and serving.
- KubeRay simplifies the management of Ray clusters and applications on Kubernetes.
- Developers use Ray cluster for interactive development. It is also better suited for short-running jobs.
- The RayJob Custom Resource Definition helps you run Ray applications for batch operations in a standalone Ray cluster.
- Ray offers Data, Train, Tune, and Serve libraries for data analytics, model training, hyperparameter optimization, and model serving respectively.
- Ray Tune integrates with MLflow (and other experiment tracking tools like Weights and Biases) for logging trials.

7

Operationalizing ML models with Kubernetes

This chapter covers:

- Fundamentals of model serving applications
- Packaging inference applications using containerization
- Best practices for serving models with Kubernetes

The life of a machine learning model and that of a student have many commonalities. Like a student, an ML model crunches through vast amounts of data to absorb intricate patterns and subtle nuances. By processing an enormous volume of information, it gradually refines its understanding and improving its predictive skills.

Ultimately, the phase of academic learning comes to an end and the student enters the workforce. Similarly, once trained, the machine learning model is ready to be deployed and ready to handle real world problems. This is the critical juncture where the proverbial rubber meets the road, transforming the model's theoretical potential into practical value. Among all the factors that dictate the efficacy and usability of a model, inference perhaps matters the most. *Inference*, the process by which a trained model makes predictions or decisions on new, unseen data, is the culmination of the machine learning pipeline. It's during inference that the model's true capabilities are put to the test in real-world scenarios. You can meticulously curate the best, most comprehensive training data, implement the most innovative and novel approaches, but if your model's real-world performance unexpectedly fails or falters in any way during inference, all the preceding hard work and ingenuity may be rendered futile.

We build machine learning systems to solve concrete business challenges. However, if our models cannot keep up with the demands of the business, our efforts will be futile. Therefore, when adding machine learning capabilities to a system, it is necessary to carefully consider non-functional requirements such as scalability, reliability, and resilience. In this chapter, we explore Kubernetes

design patterns for operating highly available and fault tolerant inference workloads.

7.1 Packaging a machine learning model

Once you've trained and fine-tuned a model, the result is saved as a serialized package. Model hubs, such as the Hugging Face Hub or PyTorch Hub, function as repositories for these serialized models. When you download a model from such a hub, you are, in essence, downloading this serialized package. A *serialized model package* is a deployable asset, ready to be integrated into a larger system.

Containerization further enhances the portability and ease of distribution of these serialized models. Encapsulating the model within a container, along with its dependencies, creates a self-contained unit that can be deployed across a variety of environments with a high degree of consistency.

Serialization is the process of converting a data structure or object, such as a trained machine learning model, into a format that can be easily stored on disk, transmitted over the network, and later reconstructed. Serialization preserves the model's architecture, learned parameters, and sometimes even its entire computational graph in a form that can be efficiently saved to disk or transferred across the network. Common serialization formats include framework-specific options like PyTorch's .pt files or TensorFlow's SavedModel format, as well as more universal standards like ONNX (Open Neural Network Exchange).

Each format offers different trade-offs in terms of compatibility, file size, and the level of information preserved. For instance, ONNX is designed to enable interoperability between different deep learning frameworks, allowing models trained in one framework to be deployed in another with minimal friction.

Let's assume you have a trained PyTorch model that has been serialized (saved) using `torch.save()`. When the time comes to serve this model, you'll load the model using `torch.load()`.

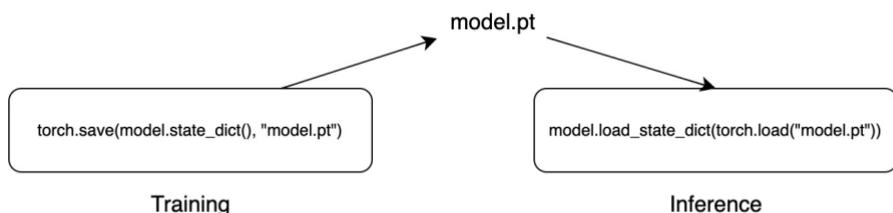


Figure 7.1 During training, the model's state dictionary is saved to a file. The saved state is

loaded back during inference.

To serve a model, you'd create an inference script. The inference script creates a pipeline that accepts input data, runs it through one or more ML models, and returns model predictions. The pipeline loads the serialized model and processes incoming requests. Inference scripts are designed to preprocess input before feeding it into the model. This might include tasks like resizing an image, tokenizing text, or normalizing numerical values. The preprocessed data is then sent to the model for predictions. The output from the model is then converted into a suitable response format like JSON.

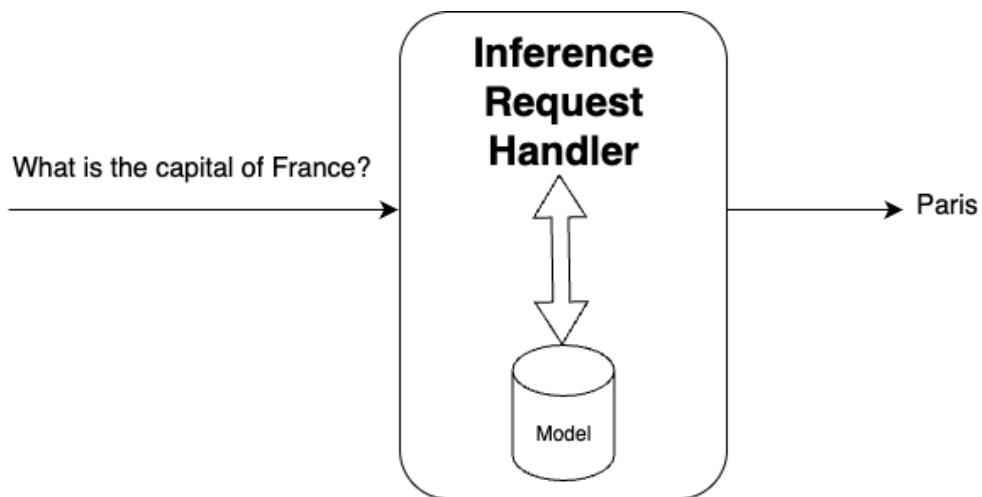


Figure 7.2 The inference pipeline receives the input, preprocesses it, runs it through the model, and sends the output back to the requestor.

There are three main components of an inference pipeline:

11. Data preprocessing – In this step the input data is preprocessed or converted to a format that the model accepts.
12. Model inference – The preprocessed data is fed into a trained model.
13. Postprocessing – The model's output is processed and converted into another format such as JSON or text.

For instance, the inference pipeline for MNIST handwritten digit recognition consists of three stages. In the preprocessing stage, the input image (a handwritten digit that we want the model to identify) is resized, normalized, and

flattened into a format suitable for the model. During model inference, this preprocessed data is fed through the neural network, which applies learned weights and activation functions to produce raw probabilities for each digit. Finally, in the post-processing stage, these probabilities are interpreted to determine the most likely digit, often with an associated confidence level. The pipeline then outputs a human-readable result, such as "The recognized digit is 7 with 98% confidence."

One of the most common ways of serving a model is over the network. In such cases, the inference script listens for connections much like a web server, waiting to receive input data, run predictions, and return results. A popular method is to wrap the model in a REST API using a web framework like Flask or FastAPI. This allows the model to be accessed via HTTP requests.

What is the capital of France?

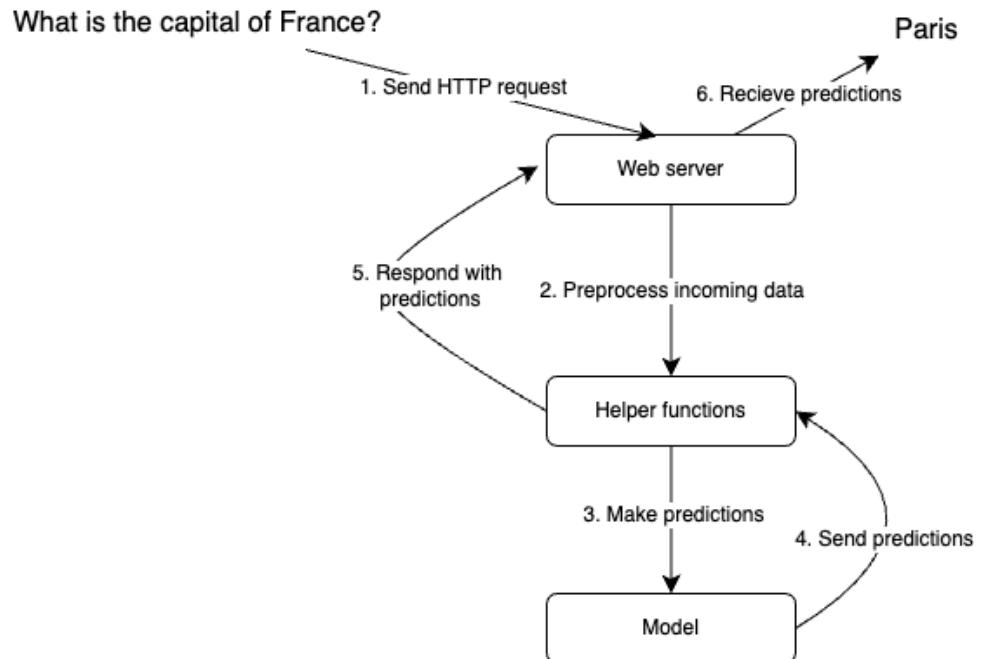


Figure 7.3 An illustration of an inference pipeline of an application that serves a model as an API. Clients use REST or a Remote Procedure Call (RPC) to invoke the model.

A variety of libraries, tools, and frameworks exist to simplify serving ML models. In Chapter 6, you learned about Ray Serve, which is one such library. PyTorch and TensorFlow also have their own methods for serving models such as

TorchServe and TensorFlow Serving respectively. Each of these solutions has their own strengths and use cases. Ray Serve, for instance, is framework-agnostic and focuses on model composition, allowing you to serve models from different frameworks together. TorchServe is specifically designed for PyTorch models, offering high performance and ease of use. TensorFlow Serving, on the other hand, is optimized for TensorFlow models and provides features like model versioning and automatic batching. Ultimately, the solution you choose will depend on your specific requirements, the frameworks you're using, and your personal preferences.

Consider the following characteristics when choosing a serving solution:

- Framework – Start with the most appropriate solution offered by your ML library.
- Scalability – If you'll need to run multiple replicas (to handle traffic) choose a framework that supports autoscaling natively.
- Capabilities – Consider the capabilities of your application. For instance, do you need model composition?
- Observability – Choose a solution that offers robust monitoring, tracing, and logging capabilities to track model performance and diagnose issues.
- Deployment options – Some frameworks support blue/green and canary deployment options that help avoiding downtime when deploying updates to the application.
- Request batching – Look for solutions that support dynamic batching to improve throughput and efficiency, especially for GPU-accelerated models.
- Latency – Consider the response time requirements of your application. Some frameworks are optimized for low-latency inference.
- Integration – Assess how well the serving solution integrates with your existing MLOps stack and infrastructure.
- Ease of use – Consider the learning curve and developer experience associated with each framework.
- Community support – A strong community can provide valuable resources, documentation, and help in troubleshooting issues.

KServe (part of Kubeflow), Seldon Core, BentoML Yatai, and Ray Serve are all popular open-source, cloud-agnostic tools for serving models on Kubernetes. Each comes with its own strengths depending on the use case. At the time of writing, Ray Serve stands out as one of the more mature and widely adopted frameworks, with usage reported at AI/ML behemoths like OpenAI, Meta, and Amazon (<https://www.anyscale.com/blog/four-reasons-why-leading->

companies-are-betting-on-ray). While the best choice will depend on your infrastructure and workflow needs, Ray's maturity, active development, and production usage in large-scale environments make it a compelling option to consider.

7.2 Why pick Kubernetes for serving models?

The most common way of serving a trained model is wrapping it into a service that can be invoked over the network using REST or gRPC API. Model serving is fundamentally similar to running typical web applications, an area where Kubernetes excels.

Since most ML models are served over HTTP, their deployment closely resembles that of traditional web applications. This similarity allows organizations to leverage Kubernetes' well-established patterns and diverse community support for web application deployment when serving ML models. Kubernetes provides essential features such as horizontal scaling, load balancing, support for various deployment strategies, which are crucial for both web applications and ML model serving.

AUTOSCALING

Horizontal scaling is particularly important for ML model serving, as it allows the system to handle varying levels of inference requests efficiently. Using Horizontal Pod Autoscaling (<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>) Kubernetes can automatically scale the number of model serving instances up or down based on demand, ensuring optimal resource utilization and responsiveness. This elasticity is vital for ML applications that may experience sudden spikes in traffic or require consistent low-latency responses.

By monitoring key performance indicators such as CPU usage, memory consumption, and request latency, Kubernetes can make informed decisions about when to scale the model serving infrastructure. This data-driven approach to scaling ensures that the system remains responsive and cost-effective, automatically adjusting to changing demands.

While Horizontal Pod Autoscaling (HPA) in Kubernetes is useful for scaling based on CPU and memory metrics, it has limitations when it comes to more complex scaling scenarios, especially for ML model serving. Request or latency-based scaling is often the most suitable approach for inference workloads. This is because ML models can only serve a limited number of concurrent requests without slowing down. When deploying a model, you should benchmark the number of concurrent requests one instance of your model can handle before slowing down or timing out. Scaling based on the number of incoming request or

response latency allows you to maintain consistent performance as traffic increases.

As mentioned in Chapter 6, frameworks like Ray have built-in support for scaling inference workloads based on requests or latency. Should you require a different metric to scale, you can also utilize KEDA (Kubernetes Event-Driven Autoscaler), which offers more sophisticated and flexible scaling options.

KEDA is an open-source Kubernetes-based event-driven autoscaler that extends the capabilities of HPA. It allows for scaling based on event sources and custom metrics, which is particularly beneficial for ML model serving. While HPA is limited to CPU and memory metrics by default, KEDA supports a wide range of custom metrics. This allows for scaling based on more relevant indicators for ML workloads, such as request latency, queue length, or even custom application-specific metrics. For ML model serving, KEDA can be particularly effective in scenarios such as:

- Scaling based on the number of inference requests per replica.
- Scaling based on the number of inference requests in a queue.
- Adjusting the number of model replicas based on prediction latency.
- Scaling different models independently based on their specific usage patterns.
- Handling burst traffic more efficiently by rapidly scaling up when needed.

If you're using a framework like Ray that provides out-of-box scaling capabilities, you'll likely not need KEDA. Other tools like Seldon Core use KEDA to drive autoscaling.

TRAFFIC DISTRIBUTION

Another aspect where Kubernetes shines is load balancing. By distributing incoming requests across multiple instances of the model server, Kubernetes ensures even utilization of resources and prevents any individual instance from becoming a performance bottleneck. This is especially important for high-traffic ML applications that need to maintain consistent performance under varying loads.

As we saw in Chapter 6, Ray Serve creates a Kubernetes Service to provide a stable IP address and DNS name that we can use to send API requests to an ML model. In our deployment, this Service was of ClusterIP type. As a result, we it was only accessible from within the cluster. We had to set up port forwarding to connect to this Service from our machines. By adding a load balancer or Ingress to this Service, we can call it from outside the cluster.

In addition, we can use Kubernetes Probes (<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>) to continually determine the health of each model server. If a model server instance becomes unhealthy or unresponsive, we can configure Kubernetes to stop sending it any additional traffic or restart the instance if it fails.

ROLLOUT STRATEGIES

Kubernetes provides robust mechanisms for updating workloads and managing their lifecycle. Features like rolling updates (<https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/>) and canary deployments allow for seamless updates of model versions without downtime, enabling continuous improvement of ML applications.

Kubernetes Ingress also makes it easy to distribute traffic between multiple versions of a model, making it easy to serve multiple versions of a model simultaneously. By implementing API versioning, you can effectively manage the lifecycle of your ML model APIs, ensuring a smooth experience for your users while allowing for continuous improvement and innovation in your ML services.

These Kubernetes capabilities enable key MLOps practices such as:

- Blue/Green deployments – Deploy new version of model alongside the older version and switch traffic between them. This enables you to quickly rollback if issues arise with the newer model version.
- A/B testing – Compare the performance of different model versions by splitting traffic between them and analyzing the results.
- Canary deployments – Gradual roll out of a new model version by directing a small percentage of traffic to it. This allows you to monitor performance of the new version before increasing its traffic distribution.

Depending on your Ingress controller (the Nginx Ingress controller is used in this book since it works in any Kubernetes environment, hybrid or cloud), you'll have various traffic distribution strategies at your disposal. These include:

- Random distribution – Route traffic based on a proportion (90:10, 50:50).
- Header-based – Route traffic based on the incoming request header.
- Path-based – Route traffic based on path. For example, "mydomain.com/v1" and "mydomain.com/v2"
- Host-based – Route traffic based on the host. For example, "v1.mydomain.com" and "v2.mydomain.com".
- Cookie-based – Use cookies to route traffic. This is typically used to

maintain session affinity, ensuring a particular user always interacts with the same model version.

With Kubernetes, you can easily update routing rules by updating your Ingress specification. You don't have to deal with load balancers, domain name servers, or other networking resources at all. Kubernetes handles that on your behalf.

You can also independently scale model versions based on its specific resource requirements and traffic load.

EFFICIENT RESOURCE UTILIZATION

Kubernetes excels in resource management, offering mechanisms to optimize the utilization of cluster resources without requiring constant manual intervention. One of its key strengths lies in its ability to efficiently bin-pack multiple workloads, maximizing resource utilization and minimizing waste.

Kubernetes intelligently schedules workloads by considering factors such as resource requirements, node capacity, and current utilization to determine the schedule workloads efficiently. It ensures that resources are distributed effectively across the cluster, avoiding scenarios where some nodes are overloaded while others remain underutilized.

For machine learning workloads, which often have specific resource requirements such as GPUs, Kubernetes can intelligently co-locate multiple models on the same node when appropriate. This bin-packing capability is particularly valuable for organizations running diverse ML workloads with varying resource needs. By efficiently packing these workloads, Kubernetes can significantly improve cluster utilization, potentially reducing the overall number of nodes required and thereby lowering infrastructure costs.

OBSERVABILITY

Observability is much vast topic that deserves in-depth exploration, which will be covered in greater detail later in the book. By standardizing how workloads emit telemetry, Kubernetes provides a consistent framework for collecting observability data (logs, metrics, and traces) that significantly enhances its value for ML operations. This standardization helps you develop software without worrying about environmental parity. Whether your Kubernetes cluster is running in AWS, Azure, Google Cloud, or on-premises infrastructure, the methods for collecting and accessing observability data remain consistent.

Developers can focus on building application features without worrying about the underlying observability infrastructure. The standardized approach means that developers can implement logging, metrics, and tracing in a consistent

manner, regardless of where the application will ultimately be deployed. This separation of concerns allows for more efficient development cycles and reduces the cognitive load on development teams.

The standardization of observability data collection in Kubernetes has fostered a vibrant ecosystem of tools and platforms. This ecosystem includes open-source solutions like Prometheus, Grafana, and Jaeger, as well as proprietary offerings from companies like Datadog, Splunk, and New Relic. The consistency in data collection methods means that these tools can easily integrate with Kubernetes environments, providing advanced features such as:

- Real-time monitoring and alerting
- Advanced data visualization
- Anomaly detection
- Performance optimization recommendations

The standardized approach allows organizations to choose observability tools that best fit their needs without being constrained by application-level implementation details. You can switch between different observability platforms or use multiple tools simultaneously without modifying your application code. For example, you could use Prometheus for metrics, Elasticsearch for logs, and Jaeger for distributed tracing, all working seamlessly with your Kubernetes-deployed applications.

The consistency in observability data collection across environments significantly enhances troubleshooting and debugging processes. MLOps teams can use the same tools and techniques to investigate issues regardless of the environment, leading to faster problem resolution and reduced downtime.

7.3 Containerizing a serve application

The first step towards deploying an ML model on Kubernetes is containerizing it. In Chapter 6, when we ran a Ray Serve application we didn't have to create a custom container image. Instead, we used the `rayproject/ray:2.34.0` image from DockerHub. And since Ray Serve supports downloading code from a URL, we simply pointed it to our code archive on GitHub. Moreover, we fetched the serialized model directly Hugging Face Hub at startup. This method allowed us to serve a model without having to create a custom container image.

There are several advantages of this approach:

- Not having to maintain a custom image eliminates time-consuming steps in the deployment pipeline. Continuous integration and deployment processes become simpler without the need to build and push custom

images for each change.

- Using a pre-built image allows for quick updates to the serving code without rebuilding containers. Changes can be made by simply updating the code archive, enabling rapid iteration and experimentation.
- By downloading the model from Hugging Face Hub at runtime, we avoided increasing the size of the container image. This improves container startup time (even though it may not be ready to serve traffic until the model bits are downloaded and loaded).
- This method separates the serving infrastructure (Ray Serve) from the application code and model. We can update each of these components independently, which improves maintainability.

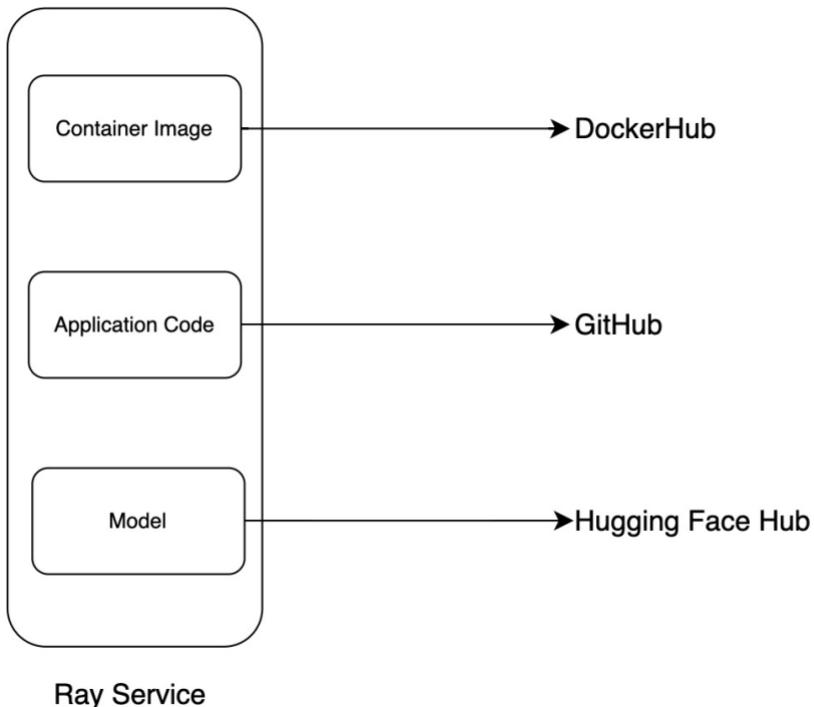


Figure 7.4 The container image, inference pipeline code, and the serialized model are the three components of a Ray Service. These components are stored in their respective repositories.

One disadvantage of this method is its excessive reliance on ultra-fast connectivity to the internet. On networks with slower or unreliable internet connectivity, it can take several minutes or even hours to download larger models, which not only makes it painful to test application or model changes but also slows down horizontal scaling. Downloading large models and code at startup can significantly increase the time it takes for the container to become operational. This delay can be particularly problematic in scenarios requiring rapid scaling or quick recovery from failures. For example, imagine a surge in traffic causing the cluster to spin up additional pods, each of those pods will have to download the model from Hugging Face before they can serve traffic. If the model is several gigabytes in size and the network is slow or throttled, this can introduce unacceptable latency and even lead to request timeouts or autoscaling failures.

There are two ways to address this concern:

- 14.Instead of downloading the code and model from GitHub and Hugging Face Hub respectively every time, we can cache the model closer to the Kubernetes cluster to reduce the dependency on the internet connection.
- 15.To eliminate dependencies on external systems, we can bake the model bits into a custom container image. The benefit of this approach is that the container image includes everything the inference application requires. The downside of this approach is that it significantly increases the size of the container image, slowing down container startup.

There are pros and cons to each approach. Building a new image every time there's an update to the model or code can become cumbersome and expensive. For this reason, the second method is usually avoided. Most teams start with using standard framework images, such as the Ray image we utilized in Chapter 6. This approach offers consistency, community support, and regular updates, making it a preferred choice for many ML teams.

Baking model and application code into the container image should be reserved for environments with unreliable or slow network connectivity. Scenarios where this approach is beneficial are:

- Edge computing scenarios where connectivity may be poor, making it essential to have all dependencies available locally.
- Air-gapped environments where internet access is restricted or entirely unavailable, necessitating self-contained images.
- Deployments in remote locations or regions with inconsistent internet access, ensuring that the application can run reliably regardless of external conditions.

- Regulatory or compliance requirements that necessitate having all code and dependencies within a single artifact for better control and auditability.
- Applications that require tight coupling between components, ensuring compatibility of the application code and model to avoid runtime issues.

Reserve packaging code, model, and dependencies for the abovementioned scenarios because they increase the operational burden on MLOps/DevOps teams. If you can get away with decoupling the model and code from the container image, prefer doing so.

7.3.1 Storing trained models

If you choose not to containerize trained ML models, you'll need a repository to store the models in a serialized format. This repository provides an intermediate storage for models. You place trained models here. At deployment time, you can instruct your inference application to download the model from this repository.

ML models, especially deep learning models, are generally very large in size. They can range from hundreds of megabytes to several gigabytes. This size consideration is crucial when choosing a storage solution for serialized ML models. If your inference application downloads the model at startup, the size of the model and network connection speed will impact the time it takes for the application to get ready during deployment and scaling. For instance, if it takes 10 minutes to download the model, your application will experience significant startup delays, which can be problematic in scenarios requiring rapid scaling or quick recovery from failures. These delays can lead to degraded performance during traffic spikes as the system struggles to scale quickly.

Long startup times also lead to inefficiencies in the system. During the download period, compute resources are underutilized, which leads to inefficient use of infrastructure. This results in increased costs, especially in cloud environments where you're billed for resources based on usage time.

To mitigate this type of slow startup, you must select a repository from where you can download models without running into network bottlenecks. Cloud-based object storage services like Amazon S3, Google Cloud Storage, or Azure Blob Storage are popular choices for storing serialized models. They provide low-cost storage and high throughput capabilities, making them suitable for storing and retrieving large ML models.

When these services are unavailable, other network file storage services come in handy. In such environments, you can load the model from a Persistent Volume that's backed by NFS or other similar networked-based storage services.

There are also dedicated tools, for example, MLflow Model Registry (<https://mlflow.org/docs/latest/model-registry.html>), that let you store and

retrieve serialized models using API. These tools also implement model versioning, which is useful for managing multiple iterations of your models and tracking their performance over time. They help you manage lifecycle of a model, from development to deployment and retirement.

7.3.2 Handling dependencies

Besides the code and model, your inference application may also need additional libraries. For instance, when we used Ray Serve to serve a model in Chapter 6, our application needed PyTorch libraries. Thankfully, all we needed to do was include these libraries in the Ray Serve configuration. When we deployed the Ray Serve application, Ray installed these libraries at startup.

The downside of this approach is that it slows down application startup. In our tests, the system took a couple of minutes to download and install these libraries. While this may not sound too terrible, during production deployments, especially in environments that require rapid scaling or quick recovery from failures, these delays can be problematic.

We can mitigate this issue by creating a custom container image that includes application dependencies. This method eliminates the need for runtime installations. It reduces startup time as all necessary dependencies are already available.

A common practice is to declare dependencies in requirements.txt file and co-locating it with application code. This way the requirements.txt file can be version-controlled along with your code. When recreating environments for testing and deployment, you maintain consistency by installing dependencies from this file. We highly recommend pinning versions in the requirements.txt file to ensure environmental consistency. You can generate this file from an existing working environment using `pip freeze > requirements.txt`, which captures the exact versions of all installed packages. Listing 7.1 shows a sample requirements.txt file.

Listing 7.1 The requirements.txt file for a Flask-based PyTorch model server

```
# Core ML libraries
numpy==1.21.0
pandas==1.3.0
torch==1.9.0

# Model serving
flask==2.0.1
gunicorn==20.1.0

# Utilities
```

```
tqdm==4.61.1  
pyyaml==5.4.1
```

Once you have a requirements.txt file, recreating the environment becomes straightforward. You can install all dependencies using pip with a single command:

```
$ pip install -r requirements.txt
```

When creating a container image, you can use the requirements.txt file to install libraries during the build process. Listing 7.2 shows a Dockerfile that incorporates this approach.

This Dockerfile performs the following steps:

- Uses the Ray project's official image as a base
- Sets the working directory to /app
- Copies the requirements.txt file
- Installs dependencies using pip
- Copies the rest of the application code (this step is optional)
- Specifies the command to run the main script

Listing 7.2 A Dockerfile

```
FROM rayproject/ray:2.34.0  
WORKDIR /app  
COPY requirements.txt .  
RUN pip install --no-cache-dir -r requirements.txt  
COPY . .  
CMD ["python", "your_main_script.py"]
```

After building this image and pushing it to a container registry (like Amazon Elastic Container Registry, Google Container Registry, or Azure Container Registry), you can use this image to deploy a Ray Serve application. Since all necessary libraries are pre-installed in the image, the application can launch the inference script immediately without needing to install additional dependencies at runtime. This results in faster deployment and scaling of your inference application.

7.3.3 Storing inference code

If you prefer using community-provided container images and downloading inference pipeline script at startup, you'll need a reliable place to store the code. The most logical place for this is your code repository.

In Chapter 6, we zipped the code and stored it in a GitHub repository. Generally speaking, the inference code is small enough that you can download it during startup without significantly impacting application launch times.

If your code archive is large (more than 100 MB) and downloading it from Git at startup is slow, you can also cache it closer to the Kubernetes cluster using:

- Cloud-based object storage services like Amazon S3, Google Cloud Storage, or Azure Blob Storage.
- Self-hosted object storage service like MinIO (<https://github.com/minio/minio>). MinIO provides an S3-compatible API and can be deployed on-premises or in the cloud, giving you more control over your data storage.
- Network storage solutions, like NFS.

7.4 Best practices for serving models with Kubernetes

Let's face it - systems fail. Period. This isn't just an occasional hazard; it's the fundamental reality of production ML systems. What separates robust ML platforms from fragile ones isn't avoiding failures (impossible) but designing systems that expect and handle them gracefully.

In software design, *reliability* is a measure of how well a system performs its intended operations over a specified window of time. The reliability of a system is influenced by three key factors:

- Availability – The proportion of time when a system is operational.
- Fault tolerance – A system's ability to withstand failures and continue operating despite the presence of faults.
- Scalability – A system's capacity to handle increased load or growth in data volume without compromising performance.

Kubernetes gives you the building blocks to address all three without reinventing the wheel. Whether you use a framework like Ray, KServe, and Seldon or build a custom application to serve models, you can use Kubernetes to make your application highly available, fault-tolerant, and resilient to failures. While it can't completely eliminate bugs in your code or improve hardware stability, it can help you mitigate the impact of these failures and avoid disrupting business operations.

7.4.1 Implement high availability

Availability is a cornerstone of system reliability. *Availability* is about ensuring that a system remains accessible to its users. It quantifies the percentage of time a system was available to its users. It is calculated using the following formula:

$$\text{Availability} = \frac{\text{Uptime}}{\text{Uptime} + \text{Downtime}} * 100$$

Another way to measure availability is by calculating the mean time between failure (MTBF). MTBF can be calculated as the average (arithmetic mean) time between failures of a system.

While achieving absolute infallibility remains an elusive goal, we can significantly enhance user experience by masking failure events. The concept of high availability hinges on redundancy. In the context of Kubernetes, this principle is implemented by running multiple replicas of an application. The fundamental idea behind this strategy is straightforward: should one Pod run into failures, other Pods stand ready to maintain service continuity.

Implementing high availability is uncomplicated in Kubernetes. For instance, with Ray Serve we can deploy multiple model server replicas by simply adjusting the replica count. With the replica count set to more than one, Kubernetes, in conjunction with the KubeRay operator, ensures that no individual model server becomes a single point of failure. This approach distributes the load across multiple instances of model servers and provides resilience against individual Pod failures.

7.4.2 Configure autoscaling

Having established high availability, the next critical facet of enhancing reliability is setting up autoscaling. Autoscaling ensures that traffic surges don't overwhelm model servers. By automatically adjusting the model server replica count, you protect a replica from receiving more traffic than it can efficiently handle.

Autoscaling is fundamentally about maintaining an equilibrium between system capacity and demand. It operates on the principle that each model server has a finite capacity for request processing, beyond which performance degradation becomes inevitable. By automatically adjusting the number of model server replicas, autoscaling protects individual servers from being overwhelmed, thereby preserving system stability and maintaining consistent response times.

In Chapter 6, we noted that Ray Serve can automatically adjust the replica count based on concurrent requests. Similarly, other Kubernetes-compatible inference frameworks provide in-built autoscaling capabilities. If your framework of choice doesn't offer in-built autoscaling, you can also use Horizontal Pod Autoscaling or KEDA as mentioned earlier in this chapter.

It's imperative to note that while autoscaling provides significant benefits in terms of resource efficiency and system responsiveness, it also introduces complexity. Careful tuning of autoscaling parameters is necessary to avoid oscillations in replica count and to ensure smooth scaling behavior. Moreover, considerations such as cold start times for new replicas and the impact on stateful applications must be taken into account when designing an autoscaling strategy.

7.4.3 Withstand hardware failures

Having multiple replicas ensures that no model replica becomes a single point of failure. However, this approach doesn't account for node failures. If all model server Pods end up getting scheduled on the same worker node, failure in that node will result in bringing down the entire application.

To address this scenario, we can utilize Kubernetes topology spread constraints. Topology Spread Constraints allow us to control how pods are spread across failure domains such as nodes, zones, or regions. They offer more granular control over pod distribution, ensuring a balanced spread of pods across the cluster. By using this feature, we can ensure that our model server pods are distributed across different nodes, thus mitigating the risk of a single node failure bringing down the entire application.

Here's how we can implement topology spread constraints to Kubernetes Deployment:

1. First, we need to add a label to our model server pods to identify them uniquely. This label will be used in our anti-affinity rules.
2. Then, we'll configure pod anti-affinity in the deployment specification.

When using KubeRay's RayService for model inference, we don't manually create Deployments. But we can still implement topology spread since KubeRay's worker group spec allows us to provide a Pod template. Listing 7.3 contains a snippet of a Ray Service resource in which we've added `app=model-server` label to the worker group spec. We can later use this label to create a Topology Spread Constraint that spreads Ray worker replicas across multiple Kubernetes worker nodes. The complete YAML file is available on the book's GitHub repository.

Listing 7.3 Adding label to Ray Service workers (ray-service.antiaffinity.yaml)

```
...
  workerGroupSpecs:
    - replicas: 2
      groupName: small-group
      template:
        metadata:
          labels:
            app: model-server
      spec:
```

```
    containers:  
      - name: ray-worker
```

...
After adding this label, we can create an anti-affinity rule to ensure that Pods with this label don't get scheduled on the same worker node. Anti-affinity rules set constraints that prevents Pods from being scheduled on the same worker node as other specified Pods. This rule is illustrated in listing 7.4.

Listing 7.4 Ray Service worker group with topology spread constraints

```
...  
  workerGroupSpecs:  
  ...  
    spec:  
      topologySpreadConstraints:  
        - maxSkew: 1 #A  
          minDomains: 2 #B  
          topologyKey: kubernetes.io/hostname #C  
          whenUnsatisfiable: DoNotSchedule #D  
          labelSelector: #E  
          matchLabels:  
            app: model-server
```

#A This specifies the degree to which pods may be unevenly distributed. A value of 1 means the difference between the number of pods on any two nodes should not be greater than 1.

#B This tells Kubernetes to have a minimum of two distinct domains. This field is optional but recommended when you want to implement a minimum spread.

#C The spread is applied at node level

#D If the constraint cannot be satisfied, the Pod will not get scheduled. The alternative is using ScheduleAnyway, which tells the scheduler to still schedule it while prioritizing nodes that minimize the skew.

#E The spread applies to any Pods with this label

This configuration tells Kubernetes to spread the model-server Pods as evenly as possible across nodes, with no more than 1 Pod difference between any two nodes. If your cluster only has 1 node and you want to schedule 2 replicas, only one replica will get scheduled. This is because we've set minDomains to 2. This forces Kubernetes to spread model server Pods on at least 2 nodes. The second replica will only get scheduled when the cluster has a minimum of 2 nodes.

Topology spread constraints are also useful when you'd like to spread replicas across multiple cloud availability zones or data centers. In the cloud, this is highly recommended. This configuration can keep your application working even if an entire availability zone becomes unavailable. To implement it, set the topologyKey to topology.kubernetes.io/zone.

7.4.4 Prevent disruption during system maintenance

In the course of maintaining a Kubernetes cluster, it becomes inevitable that one must periodically engage in the maintenance of worker nodes. Such maintenance activities may include rebooting, updating, patching, or even replacing these nodes. To ensure the continuity of services and to minimize potential disruptions, it is considered a best practice to systematically remove all running Pods from a node before undertaking these maintenance tasks. This process of methodically evacuating all Pods from a node is referred to as "*draining* a node." It is performed by running `kubectl drain <node name>`.

When a node is drained, all Pods currently running on the targeted node are evicted and subsequently terminated. However, it is crucial to note that this process is not carried out in a vacuum. To maintain service integrity and minimize disruptions, Kubernetes takes a preventative measure. Prior to the termination of existing Pods, Kubernetes creates replacement Pods, which are then scheduled on other available nodes within the cluster.

This approach ensures a seamless transition of workloads from the node undergoing maintenance to other operational nodes in the cluster. The replacement Pods are created and brought to a ready state before the original Pods are terminated, thereby maintaining the desired replica count and preserving the application's availability throughout the draining process.

During this procedure, it is possible that multiple replicas belonging to a particular workload get impacted. For instance, consider a scenario where you have an inference application with three model server replicas, and two of these replicas happen to be running on the node scheduled for maintenance. In such a case, the draining process will affect a majority of the application's instances simultaneously, which may lead to service degradation.

To mitigate the potential risks associated with such situations, you can create a Pod Disruption Budget (PDB) for the inference application. A *PDB* limits the number of Pods that can be taken down simultaneously during planned node maintenance events. PDB prevents the simultaneous eviction of too many Pods from the same application. For example, a PDB might require that at least 50% of the Pods for a given application must always remain available. In our scenario with three replicas, this would mean that Kubernetes would only evict one Pod at a time, scheduling the creation of its replacement before proceeding with the eviction of the second Pod.

Listing 7.5 shows a PDB that applies to any Pods with label `app=model-server`. This PDB tells Kubernetes to prevent having a situation where less than 3 Pods are available. If this cannot be ensured, Kubernetes will prevent draining the node that's running one or more of these Pods.

Listing 7.5 PDB for model server Pods (pdb-ray-service.yaml)

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: ray-service-pdb
spec:
  minAvailable: 3 #A
  selector:
    matchLabels:
      app: model-server
#A The minimum number of Pods that Kubernetes should keep available.
```

Before initiating the node draining procedure, it is important to assess the cluster's capacity. Ensure the cluster has enough capacity before draining a node. The Pods currently running on the targeted node will need to be rescheduled on other nodes. If your cluster can't fit these Pods on other nodes in your cluster (or if your cluster doesn't autoscale), then it will keep a node from getting drained.

7.4.5 Continuous health assessment

Self-healing and failure management are the cornerstones of a reliable system. In modern distributed systems, the ability to automatically detect, diagnose, and recover from failure is not only a desirable feature but also an absolute necessity. As systems grow larger, their complexity increases, and so does the likelihood of component failures.

The concept of self-healing in distributed systems draws inspiration from biological systems, which have evolved sophisticated mechanisms to maintain homeostasis and recover from injuries. In software architecture, self-healing refers to a system's ability to autonomously identify deviations from its normal operational state and take corrective actions without human intervention.

In Chapter 2, we touched upon Kubernetes probes and how they help you detect failure in applications and provide a mechanism for automatic remediation. They are essential to improve the reliability and resilience of your applications, including model servers.

To recap, Readiness probes determine when Pod is ready to receive traffic. In contrast, Liveness probes check whether an application is alive and responsive.

Frameworks like Ray Serve automatically configure Liveness and Readiness probes. For instance, Ray Serve creates a Readiness probe that sends an HTTP request to the model server (Ray workers) to check if it is ready to accept traffic. When the model server successfully responds to the request, Kubernetes knows it is ready. Listing 7.6 contains a Readiness probe as created by KubeRay.

Listing 7.6 Ray Serve Readiness Probe

```
readinessProbe:
```

```

exec:
  command: #A
  - bash
  - -c
  - wget -T 2 -q -O-
http://localhost:52365/api/local_raylet_healthz | grep success
&& wget -T 2 -q -O- http://localhost:8000/-/healthz | grep
success
  failureThreshold: 1 #B
  initialDelaySeconds: 10 #C
  periodSeconds: 5 #D
  successThreshold: 1 #E
  timeoutSeconds: 1 #F
#A The commands to be executed
#B The probe will be considered failed after 1 consecutive failure
#C Kubernetes will wait 10 seconds after the container starts before running the first
probe
#D The probe runs every 5 seconds
#E The probe is considered successful after 1 successful attempt
#F The probe has 1 second to complete before it's considered failed

```

If the Readiness probe fails, Kubernetes stops sending it any traffic.

Kubernetes also continues to probe the Pod's health using a Liveness probe. This probe continuously sends HTTP requests to the model server. Should the server fail to respond to the request, Kubernetes terminates the Pod, and KubeRay creates another Pod to compensate for the loss. Listing 7.7 contains a Liveness probe.

Listing 7.7 Ray Serve Liveness probe

```

livenessProbe:
  exec:
    command:
    - bash
    - -c
    - wget -T 2 -q -O-
http://localhost:52365/api/local_raylet_healthz | grep success
  failureThreshold: 120
  initialDelaySeconds: 30
  periodSeconds: 5
  successThreshold: 1
  timeoutSeconds: 1

```

It should be noted that misconfigured probes can lead to false negatives, impacting the availability of the application. This is one of the reasons for setting the `failureThreshold` of the Liveness probe to 120 seconds. It ensures that fleeting issues that might cause the probe to fail don't lead to unnecessary Pod terminations. With `periodSeconds` set to 5, the Pod will only be terminated if its Liveness probes fail for 300 seconds ($120 * 30$) consecutively.

When using a model serving framework, it is likely that you won't spend too much time configuring or customizing these probes. In other cases, you can indeed utilize probes to improve the availability of your systems.

7.5 Model Version and Lifecycle Management

Over time, models evolve, either through retraining, fine-tuning, or architecture changes, and the MLOps system must support rolling out, testing, and, if needed, rolling back these versions in a controlled manner.

Following DevOps best practices, developers store ML application code in a version-controlled code repository. CI jobs are triggered on code changes, performing tasks such as running unit tests, packaging model artifacts, and building container images. These CI pipelines produce deployable assets, typically container images tagged with a semantic version or commit SHA, and optionally upload serialized model files to an object store or model registry.

Once the model and its associated inference code are packaged, the deployment step hands off control to Kubernetes. At this point, the CI system either:

- Directly applies Kubernetes manifests to the cluster (for example, `kubectl apply -f deployment.yaml`), or
- Pushes manifest changes to a Git repository watched by a GitOps controller (like ArgoCD or Flux).

In both approaches, the goal is to ensure that any change to a model or its code is deployed reproducibly, traceably, and safely. In GitOps, Git becomes the source of truth, not just for application code, but also for infrastructure state. This is particularly important in regulated environments or high-availability systems where auditability and rollback are non-negotiable.

A key advantage of using Kubernetes as the deployment target is its native support for rolling updates, health checks, and fine-grained resource control. When a new version is rolled out, Kubernetes gradually updates Pods, monitors their health via readiness and liveness probes, and automatically halts or rolls back the deployment if issues are detected. This gives teams a safety net to test new model versions under live traffic with minimal risk.

In serving scenarios involving multiple model versions, Kubernetes can help manage traffic routing between versions, whether via path-based routing (`/v1`, `/v2`), host-based routing (`v1.example.com`), or weight-based canary deployments. When combined with advanced routing techniques like traffic mirroring or request logging, this enables teams to compare model behaviors

side-by-side in production, accelerating experimentation without disrupting the user experience.

Note that model versioning begins with how you organize and store serialized models. Whether using a cloud object store like Amazon S3, a model registry like MLflow, or Git-based solutions, it is essential to encode metadata such as version numbers, timestamps, commit hashes, and training parameters. This metadata allows teams to trace back predictions to the specific model version used, which is invaluable for debugging and compliance. A few best practices:

- Use semantic versioning for models: my-model:v1, v1.1, v2-beta.
- Store models in versioned paths in your storage backend (s3://models/my-model/v1/model.pt) or use tags if using MLflow or Hugging Face Hub.
- Embed the model version into the API URL (/v1/predict, /v2/predict) or HTTP headers so clients can choose which version to invoke.

When it comes to serving multiple versions, Kubernetes offers first-class support. You can:

- Deploy each model version as a separate service or Deployment, optionally using traffic-splitting strategies via Ingress (as described in Section 1.2).
- Use Ray Serve or KServe's built-in versioning: Ray Serve, for example, allows you to tag deployments with version metadata and swap traffic with a single update to the ServeDeployment config.
- Leverage blue/green or canary deployments to progressively shift traffic and monitor performance before fully switching to a new version.

Kubernetes Container lifecycle hooks (<https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/>) can also be embedded in your Deployment strategy:

- Post-start hooks to verify the model has loaded correctly.
- Readiness probes to indicate that a newly deployed version is actually ready to serve.
- Pre-stop hooks to gracefully drain requests from an old version before shutdown.

Finally, you need to define deprecation and rollback policies. If a model underperforms in production, automated rollback (via metrics thresholds) or manual re-promotion of the previous version should be part of the operational plan. In practice, many teams retain the last N versions in production with active monitoring dashboards for live comparison using business-specific KPIs.

Model versioning is not a one-time event, it's a continuous process tied to experimentation, deployment, and business outcomes. The more repeatable and observable this process is, the faster you can innovate without compromising reliability. Ultimately, lifecycle management is about confidence: the confidence to deploy frequently, test rigorously, and recover quickly. Kubernetes, paired with a disciplined CI/CD strategy, provides the operational scaffolding needed to deliver ML models at the pace of modern software development.

7.6 GPU sharing

Of all the hardware components that a machine learning application requires, GPUs are the most expensive. By default, when you allocate a GPU to an application, the entire GPU is dedicated to the process. For instance, consider your worker node has an NVIDIA A100 GPU. This GPU comes with 40 GB memory. If your ML application doesn't consistently demand the full capacity of this GPU, you can improve utilization by implementing GPU-sharing techniques.

While GPU sharing isn't unique to Kubernetes, it is certainly easier to configure nodes to enable GPU sharing and have multiple processes (Pods) share the same GPU resources. NVIDIA offers multiple methods to improve GPU utilization. The three approaches that are widely implemented are:

16. Multi-instance GPU (MIG) – Partitions a GPU into smaller GPU instances
17. Multi-processing Service (MPS) – Allows multiple processes to share a single GPU concurrently
18. Time-slicing – Similar to MPS, but slower as it requires context switching

The fundamental difference between MIG and other methods is that in MIG you create partitions in advance and then allocate fractional GPUs to workloads. As a result, you need to know how to partition a GPU. In contrast, with MPS and time slicing, you configure how many processes can concurrently use a GPU.

Profile your applications and understand their GPU utilization patterns. If your models are using less than 40–50% of the GPU consistently, you're likely overpaying for idle capacity. In such cases, implementing GPU-sharing techniques can significantly reduce infrastructure costs. MIG is ideal for strong isolation and deterministic performance, making it a good fit for production serving scenarios. MPS is better suited for dynamic or experimental workloads that benefit from the flexibility of shared access. Be aware, however, that using these techniques in Kubernetes often requires custom node setup, NVIDIA's device plugin, and appropriate scheduling configurations. Despite the extra effort, the payoff in utilization efficiency and cost savings can be substantial.

The NVIDIA GPU Operator includes the DCGM exporter (<https://github.com/NVIDIA/dcgm-exporter>), which exposes GPU utilization

metrics such as memory usage, compute saturation, and temperature. These metrics can be collected via Prometheus and used to determine how efficiently your models are using GPU resources. Later in the book, we'll explore GPU monitoring and observability strategies in detail, including how to visualize and act on this telemetry to improve cluster efficiency.

Summary

- Kubernetes is a robust, feature-rich platform for serving machine learning models.
- Model frameworks such as Ray Serve, KServe, and Seldon Core provide abstractions to simplify model inference.
- The three key components of a Kubernetes-based inference application are the container image, pretrained model, and inference pipeline.
- To reduce model serving startup time, store the serialized model closer to your Kubernetes cluster.
- Use horizontal scaling to deploy models with high availability and autoscaling.
- Mitigate the impact of infrastructure failure by spreading model servers across worker nodes and cloud availability zones or data centers.
- Use Pod Disruption Budgets to avoid downtime during planned infrastructure maintenance events.
- Catch failures in application with the help of Kubernetes probes.
- Profile applications and monitor GPU utilization to identify areas of improvement. For applications that don't fully utilize GPUs, consider adopting a GPU sharing strategy.

8

Serving LLMs on Kubernetes

This chapter covers:

- Understanding the basics of hosting large language models
- Serving LLMs efficiently with vLLM
- Running vLLM on Kubernetes
- Performing distributed inference with vLLM, Ray, and Kubernetes

In Chapter 7, you learned how to serve ML models on Kubernetes using Ray. While Ray provides a robust framework for model serving, it is not inherently optimized for the unique challenges posed when running Large Language Models (LLMs). A Large Language Model (LLM) is a type of deep learning model trained on vast corpora of text to understand and generate human-like language. These models are typically built using Transformer architecture ([https://en.wikipedia.org/wiki/Transformer_\(deep_learning_architecture\)](https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture))) and are capable of performing tasks such as text completion, summarization, translation, and code generation. If you've used ChatGPT, Claude, or messed around with Meta's AI tools, congrats, you've already met an LLM.

LLM inference requires specialized optimizations for efficient memory management, batching, and throughput scalability. The problem? Serving LLMs isn't as simple as throwing them into your usual model server. They're resource-hungry, need precise memory handling, and can choke your GPU if you're not batching requests efficiently. Standard model hosting approaches fall apart when hosting LLMs, too much memory usage, sluggish response times, and wasted GPU cycles. Specialized inference engines such as vLLM address these challenges by introducing features tailored for LLM inference.

In this chapter we will learn how vLLM helps you run LLMs in a Kubernetes cluster. Once you understand how vLLM works, we'll explore how it works with Ray for distributed inference and serving. By the end of this chapter, we will have an LLM running in our Kubernetes cluster. Along the way we will learn about the common optimization techniques for LLM inference workloads.

8.1 Unique challenges of serving LLMs

GPUs are expensive, and running LLMs on them can quickly become cost-prohibitive without the right optimizations. Every optimization in this space is ultimately aimed at reducing the cost of generating text, or images, with large models. Serving LLMs is fundamentally different from serving classical ML models, like a text classification model, because of their massive size, computational demands, and unique inference characteristics. To be specific, we're referring to LLMs based on the Transformer architecture introduced in Vaswani et al.'s 2017 paper "Attention Is All You Need". In this architecture, the model processes input data using self-attention mechanisms, which captures intricate relationships between words (or tokens). However, the same mechanisms that give LLMs their power also make them significantly more resource-intensive, both during training and inference. These models require so much GPU capacity that even a relatively modest-sized model with 13 billion parameters can overwhelm a single Nvidia A100 GPU.

One of the primary challenges in LLM inference is the management of the Key-Value (KV) cache. During text generation, a Transformer model stores previously computed key and value tensors for each token in GPU memory. This allows the model to efficiently reference past context without recomputing activations, drastically speeding up inference.

However, as the model generates longer outputs, the KV cache grows proportionally, consuming an increasing amount of GPU memory. For large models handling long conversations or documents, this can quickly exceed the capacity of a single GPU, leading to memory exhaustion.

To mitigate this, inference frameworks employ techniques that distribute computations and KV cache storage across multiple GPUs, and memory-efficient data structures. Without these optimizations, LLM inference would be prohibitively slow or require impractically large GPU clusters.

Another critical distinction from traditional ML models is that LLMs generate text iteratively, predicting one token at a time based on previous outputs. This autoregressive nature contrasts with models like image classifiers or text classifiers, which process an input in a single forward pass and immediately return a result. Since each new token depends on the previously generated tokens, LLM inference creates an irregular, sequential workload that can be challenging to optimize for efficient hardware utilization.

To improve inference efficiency, frameworks use *batching*, where multiple input sequences are processed simultaneously. By batching requests together, the model can maximize parallelism, reducing memory access overhead and improving overall throughput. However, even batching has limitations, different requests may require different numbers of tokens, leading to inefficiencies.

Additionally, various model compression techniques are often employed to make inference more efficient. Quantization, for instance, reduces the precision of model weights (e.g., from 16-bit floating point to 8-bit integers), decreasing memory requirements and accelerating computation with minimal accuracy loss. These optimizations are essential for making LLM inference feasible at scale, whether for real-time applications like chatbots or large-scale deployments serving thousands of concurrent users.

8.1.1 *LLM Inference Servers*

LLM inference servers are purpose-built systems designed to run large language models efficiently, often in real-time and at scale. These servers handle tasks like batching, token generation, caching, GPU memory management, and API compatibility, all with the goal of reducing latency and maximizing throughput per GPU. Because LLMs are computationally expensive and resource-intensive, inference servers play a key role in making their use practical and cost-effective.

There are several Kubernetes-compatible LLM inference servers available today, each with different strengths and trade-offs. vLLM is one of the most popular open source options, offering high throughput and excellent memory efficiency through features like paged attention, continuous batching, and support for the OpenAI-compatible API. NVIDIA NIM is a commercial alternative that packages optimized inference endpoints for NVIDIA models, offering enterprise support and integration with the NVIDIA AI ecosystem. Another option is TGI (Text Generation Inference) by Hugging Face, which provides a scalable serving stack focused on transformer models. Each of these frameworks aims to make LLM serving faster, cheaper, and easier to integrate into real-world applications.

Table 1.1 The three widely used LLM inference servers

Feature	vLLM	NVIDIA NIM	TGI (Hugging Face)
License	Open Source (Apache 2.0)	Commercial (NVIDIA AI Enterprise)	Open Source (Apache 2.0)
Optimizations	Paged Attention, Continuous Batching	TensorRT, CUDA- optimized kernels	Token streaming, batch optimization
GPU Efficiency	High	Very High	Good

		(hardware-tuned)	
OpenAI API Compatibility	Yes	Yes (proprietary APIs)	Yes
LoRA Adapter Support (more on what this means later in this chapter)	Yes (Multi-LoRA)	Limited	Yes
Kubernetes Support	Helm	Helm	Helm
Metrics and Monitoring	Provides Prometheus metrics	Provides Prometheus metrics	Provides Prometheus metrics
Ideal Use Case	Open models, custom deployments	NVIDIA models, enterprise-scale serving	Hugging Face models

Let's explore how vLLM incorporates inference optimization strategies to provide a high-performance, scalable LLM inference solution within a Kubernetes environment. Rest assured, the concepts you'll encounter in this chapter, like KV cache reuse, continuous batching, GPU scheduling, and multi-model routing, apply broadly across modern LLM inference servers. Even if you later choose to deploy a different system like Hugging Face's TGI, most of the high-level ideas and architectural patterns will remain the same. The tools may differ, but the principles of efficient LLM serving in Kubernetes carry over.

8.2 *Introducing vLLM*

vLLM (Virtual Large Language Model) is an open source LLM inference framework designed to improve throughput in LLM inference while improving memory usage and GPU utilization. It was developed at UC Berkley to address the unique challenges of LLM inference, including the need for efficient memory management, parallelization, and batching. It provides high-performance model serving with minimal overhead, making it a strong alternative to more general-purpose serving solutions.

One of the main advantages of vLLM is its ability to achieve the same (or better) LLM performance using fewer GPUs. By optimizing how requests are batched, how key-value (KV) caches are stored, and how generation steps are scheduled, vLLM extracts more work out of each GPU cycle. In practice, this means you can handle higher throughput with smaller or fewer GPU instances,

reducing infrastructure costs without sacrificing response times. This efficiency is especially important at scale, where GPU usage often represents the largest share of an ML application’s operational budget.

vLLM introduces a more efficient way to handle KV caching and request batching, two of the biggest bottlenecks in LLM inference. Unlike standard inference engines, which often suffer from inefficient GPU memory management due to static batching, vLLM implements a novel *Paged Attention* mechanism. In traditional LLM servers, each query reserves a large, contiguous chunk of GPU memory for its attention KV cache (whether it uses it fully or not), leading to fragmentation and wasted memory. Paged Attention instead breaks the KV cache into fixed-size “pages” and allocates them on-demand. Much like virtual memory paging in operating systems, this lets vLLM dynamically reuse and allocate memory only as needed.

The result is significantly reduced memory footprint and the ability to pack more (or longer) queries onto the GPU without running out of memory. Before vLLM, servers often pre-allocated memory per request (many chunks sitting unused); with Paged Attention, memory is pooled and shared, eliminating much of the KV cache waste and even enabling larger batch sizes than otherwise possible.

Another key advantage of vLLM is its ability to support *continuous batching*. Traditional batching methods struggle with handling multiple requests of varying lengths efficiently. If a batch contains sequences of different lengths, shorter sequences may finish processing earlier, leaving idle GPU capacity. vLLM’s continuous batching dynamically adjusts batch composition, allowing new requests to be added as others finish, thereby maintaining high GPU utilization. This is particularly important for LLM serving, where response times must be optimized while handling unpredictable workloads.

Additionally, vLLM integrates parallelism techniques and optimized CUDA kernels to maximize performance on modern GPUs. It is designed to take full advantage of hardware acceleration without requiring deep modifications to existing model architectures. As of writing, it supports a range of popular LLMs and vision language models, including GPT-based models, LLaMA, Falcon, and others, making it a flexible choice for self-managed LLM deployments.

Other vLLM features include:

- Multi-LoRA Support – LoRA fine-tunes only a small number of model parameters and stores them as lightweight adapters, drastically reducing the cost of training and serving custom models. vLLM allows multiple LoRA (Low-Rank Adaptation) adapters to be loaded simultaneously, making it easier to serve fine-tuned models without duplicating base weights.

- Speculative Decoding – Speeds up inference by predicting multiple tokens at once and verifying them, reducing the number of sequential generation steps where each new word depends on all previous words.
- CUDA Graph – Reduces CPU overhead and improves GPU execution efficiency by pre-recording computation graphs and reusing them during inference.
- Flash Attention – An optimized attention mechanism that significantly reduces memory bandwidth usage, leading to faster token generation.
- Flash Decoding – A technique that accelerates decoding by optimizing how tokens are processed and retrieved from memory.
- Chunked Prefills – Improves batch efficiency by breaking up long inputs into smaller chunks, allowing for better GPU utilization.
- Automatic Prefix Caching – Caches shared prompt prefixes across requests to eliminate redundant computation and speed up multi-user inference.

Even though most of vLLM's performance optimizations are geared toward serving large language models on NVIDIA GPUs, the project also supports a wide range of other hardware, including AMD CPUs and GPUs, Intel CPUs and GPUs, PowerPC CPUs, Google TPUs, and AWS Neuron devices. While you may not get the same level of throughput or feature completeness on all of these platforms, the core inference engine is designed to be flexible and extensible across different backends. This makes vLLM a viable option not only in high-performance GPU clusters, but also in edge, hybrid, or cost-sensitive environments where non-NVIDIA hardware is preferred or required.

In addition to its performance optimizations, vLLM includes a compatible implementation of the OpenAI API server. This means it can serve LLMs using the same REST interface as OpenAI's GPT models, including support for /v1/chat/completions, /v1/completions, and /v1/models endpoints. As a result, you can integrate vLLM into any existing application that relies on the OpenAI API, such as chat interfaces, tools built with LangChain, or developer plugins, without needing to rewrite your client code. This compatibility makes it easy to switch from proprietary APIs to a self-hosted LLM while maintaining full control over the model, infrastructure, and data.

8.2.1 vLLM Support for Kubernetes

vLLM offers strong native support for Kubernetes through its official open-source project, vllm-project/production-stack (<https://github.com/vllm-project/production-stack/>). This project provides a reference deployment of vLLM tailored for production environments, including Helm charts and manifests that

simplify the process of deploying vLLM on a Kubernetes cluster. It has best practices for scalability, GPU utilization, API compatibility, and monitoring, making it easier for teams to self-host LLMs reliably in cloud-native environments.

The production stack includes support for deploying the OpenAI-compatible API server, multi-model support, configuring models and LoRA adapters, and attaching storage. It also supports GPU resource requests and limits via native Kubernetes configurations, enabling you to control how many GPUs each replica can access. This makes it easy to right-size deployments for different models or traffic patterns, and ensures that GPU resources are allocated efficiently across your cluster.

The stack also provides out of the box observability for inference. It creates Kubernetes liveness and readiness probes for health checks, allowing a vLLM deployment to be monitored and automatically restarted if necessary.

vLLM's official Kubernetes support is especially valuable for MLOps teams that want to standardize model deployment pipelines without having to write custom infrastructure code. With just a few configuration values, you can set up a production-grade, scalable LLM inference service that plugs into your internal developer tools or external APIs.

8.2.2 vLLM Production Stack Architecture

The vLLM production stack has three core components routers, engines, and metrics. Each plays a distinct role in managing traffic, executing inference workloads, autoscaling, and monitoring the health and performance of the system.

vLLM: ROUTER

The vLLM *router* serves as the front-end for incoming client requests. It implements the OpenAI-compatible API gateway that provides a single entry point for routing requests to multiple vLLMs engines. It handles tasks such as request validation, routing, and batching coordination. Routers are stateless and can be scaled independently based on traffic volume. They do not run inference themselves, but instead forward requests to one or more backend vLLM engine instances. This separation of concerns allows the system to scale horizontally and isolate user-facing API logic from model execution.

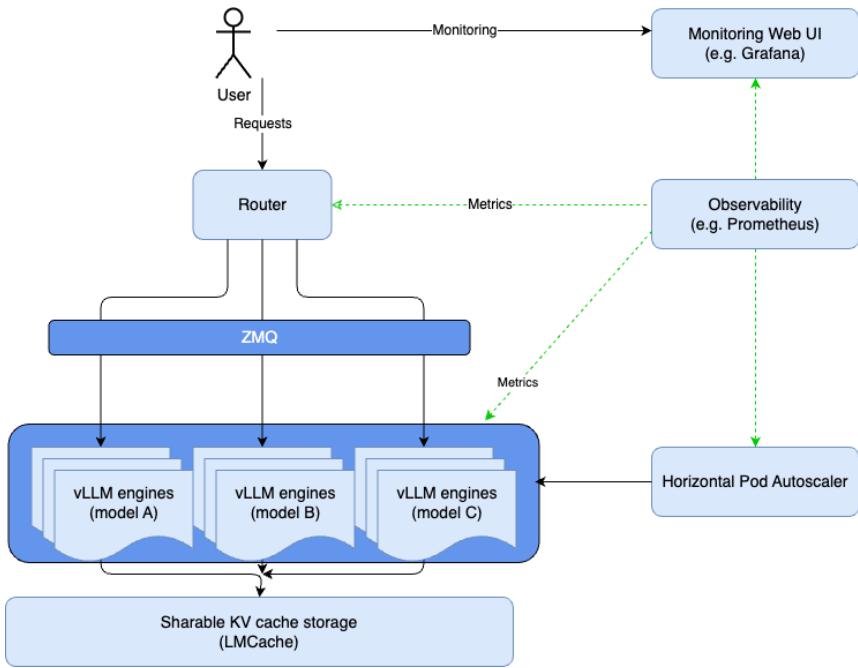


Figure 8.1 Using the vLLM router directs user requests to appropriate vLLM engine instances running different models. These models can share a common KV cache storage layer (LMCache) for improved efficiency. vLLM produces metrics provide oversight and the basis for scaling vLLM engines horizontally.

vLLM routers direct traffic to engine instances using three main strategies:

1. Round-robin routing (default) – Distributes requests evenly across all engine Pods, ideal for stateless, one-off inference requests.
2. Session-based routing – Routes all requests from the same conversation to the same engine Pod, critical for multi-turn interactions.
3. KV cache-aware routing – Routes requests based on KV cache locality, sending each request to the engine that contains the longest matching prefix in its cache. This maximizes computational reuse across related prompts.

The router maintains an in-memory session map to track which engine handles which conversation. This routing intelligence dramatically improves performance by avoiding redundant computation of context, a significant bottleneck in LLM serving.

In multi-replica deployments, each router Pod maintains its own session map. This means session-based routing will only work correctly if the same client

consistently hits the same router instance. If you place a load balancer in front of the routers (which is common), you should configure session affinity at the load balancer level to preserve KV cache locality. Most major cloud providers support this: AWS Application Load Balancer (ALB) offers sticky sessions via cookies, Google Cloud Load Balancing supports client IP-based affinity, and Azure Application Gateway includes session persistence features. Enabling one of these options ensures that all requests from a client session are consistently routed to the same router Pod, allowing vLLM to reuse the session's KV cache effectively.

Routers also handle error and exception management. For instance, if a request is malformed, targets an unsupported model, or exceeds configured limits, the router returns a clear HTTP error response to the client. In cases where a downstream engine is unavailable or encounters an inference failure, the router captures the exception and returns a standardized error message, helping clients distinguish between user input issues and system-side problems. Besides directing requests, routers ensure reliability. They protect the backend engines from invalid input, thus improving the system's robustness.

The router is built with FastAPI and Uvicorn, giving it the ability to handle high request volumes with low latency. It integrates with Kubernetes, so it can automatically scale based on traffic, distribute requests across Pods, and take advantage of features like service discovery and Pod-level health checks.

Routers and engines communicate using ZeroMQ (ZMQ), a high-performance asynchronous messaging library. This allows for low-latency, reliable communication between components, and decouples the router logic from the inference logic running on the engines. ZMQ provides an efficient transport layer for passing batched inference requests and responses, enabling the system to scale independently and maintain responsiveness even under heavy load.

vLLM: SERVING ENGINES

The second component of the stack is the *vLLM serving engine*, which is responsible for receiving requests from routers and running the model to generate outputs. The inference process includes several coordinated stages: input processing, request scheduling, model execution, and output postprocessing.

Input processing handles the tokenization of raw input text using the tokenizer configured for the model. *Tokenization* is the process of converting human-readable text into numerical tokens that the model can understand and process. This step is necessary because language models operate on sequences of token IDs rather than raw text, and different models may use different tokenization strategies depending on their architecture and training data. Scheduling determines which requests are executed at each step, balancing load

and optimizing GPU usage. Model execution is responsible for running the forward pass of the model, which may involve distributed execution across multiple GPUs if configured. Finally, output postprocessing converts the generated token IDs back into human-readable text before returning the result to the router.

Each engine instance typically loads a full model into memory and is assigned one or more GPUs. Engines support multiple models and LoRA adapters, enabling the deployment of fine-tuned variants alongside base models. KV cache is managed locally on each engine and is reused across requests in the same session to speed up generation. This is especially useful for chatbot-like applications where multiple related inputs share a common context.

vLLM also supports an optional feature called CPU offloading, where portions of the KV cache can be stored in CPU memory instead of GPU memory. This shared cache layer, provided by LMCache (<https://github.com/LMCache/LMCache>), allows cache pages to be reused across similar or repeated requests, reducing redundant computation and improving memory efficiency. In multi-GPU environments, this helps prevent redundant copies of the same prompt tokens across devices. In multi-node setups, however, the shared cache is not replicated across nodes. Each engine instance maintains its own local and offloaded CPU cache, and cache reuse is only possible within the same engine. For this reason, routing requests to the same engine instance is especially important in distributed deployments to preserve cache locality and minimize expensive recomputation. Engines can be scaled horizontally and deployed with precise GPU resource configurations for optimal placement in Kubernetes clusters.

METRICS

vLLM exposes internal metrics that can be used to monitor the health of the system. These metrics are exposed via the `/metrics` endpoint on the vLLM OpenAI-compatible API server. A cluster-level monitoring tool like Prometheus (a popular open-source monitoring system we'll dive deep into later in the book) can scrape these metrics. As a result, you can monitor vLLM deployments using your existing monitoring and alerting infrastructure.

The metrics cover various aspects of the inference workload, including request status, latency, GPU cache usage, token throughput, and engine-level activity. For example, you can track how many requests are currently running, how many are waiting in the queue, and how much of the GPU and CPU memory is being used for caching. There are also counters for the number of prompt and generation tokens processed over time, and histograms that capture how long it takes to generate the first token or complete a full request. These metrics help

you understand how efficiently the system is processing inference workloads and where potential bottlenecks may be.

These metrics are especially useful in Kubernetes environments for autoscaling. For instance, if the number of waiting requests is consistently high, it may indicate that the system is under-provisioned and needs more engine replicas. Similarly, sustained high GPU cache usage could point to the need for larger or more powerful GPU nodes. By integrating these metrics into a Horizontal Pod Autoscaler (HPA), you can dynamically scale your vLLM deployment based on real-time load, ensuring that performance remains steady as traffic increases or decreases.

8.3 Getting started with vLLM

vLLM provides two primary ways to run a language model: online inference and offline inference. You can pick the approach that best suits your workload, whether you need a continuously running service or a quick, ad hoc job.

- *Offline inference*, also known as batch inference, processes data in batches or groups, generating predictions for multiple input samples at once. This approach can handle high throughput efficiently but introduces higher latency due to the batched processing. It's well-suited for scenarios where predictions don't need to be generated immediately, such as pre-computing recommendations, generating reports, and conducting large-scale data analysis.
- *Online inference* processes data individually as it arrives, generating predictions for each input sample immediately. Low latency is critical in this mode, as predictions need to be generated quickly for real-time applications. While online inference excels at providing instant responses, it might have limitations on throughput, especially with complex models or constrained resources. Common use cases include chatbots, personalized recommendations, fraud detection systems, and real-time language translation.

NOTE To follow along with this section, use a Jupyter notebook with GPU support. This approach prevents dependency conflicts and delivers optimal performance.

The simplest way is to use the vLLM command-line interface (CLI), which launches an OpenAI-compatible server with minimal configuration. This approach is ideal for real-time or interactive use cases where you need to send requests

over HTTP. For example, to serve the “facebook/opt-125m” model using vLLM, just install the library with:

```
$ pip install vllm
```

Then start the server:

```
$ vllm serve --model facebook/opt-125m
```

With this, vLLM automatically downloads and loads the model into GPU (or CPU, if no GPU is available) memory, then listens on port 8000 for incoming requests. Depending on your network connection, this operation can take a few minutes. We can view the list of models the server currently hosts by running:

```
$ curl http://localhost:8000/v1/models
{"object": "list", "data": [{"id": "facebook/opt-125m", "object": "model", "created": 1744142465, "owned_by": "vllm", "root": "facebook/opt-125m", "parent": null, "max_model_len": 2048, "permission": [{"id": "model_permission-125m", "parent": null, "max_model_len": 2048, "allow_create_engine": false, "allow_sampling": true, "allow_logprobs": true, "allow_search_indices": false, "allow_view": true, "allow_fine_tuning": false, "organization": "*", "group": null, "is_blocking": false}]}]}
```

You’ll see a JSON response containing information about each model, such as its name. This is the same endpoint you would call on OpenAI’s service, but here it’s served locally through vLLM. Once you confirm your model is loaded and ready, you can proceed to the /v1/completions endpoint to generate text. This method is perfect for scenarios like user-facing applications, chatbots, or microservices that require a continuously running API. Once it’s up and running, you can send requests to the server using the same REST endpoints as you would for OpenAI’s GPT endpoints, for example, “POST /v1/completions”.

Behind the scenes, vLLM takes care of tokenizing your prompt, running inference on the loaded model, and returning a JSON response with the generated text. This CLI command is the quickest way to try out a model or integrate it with applications already designed to talk to an OpenAI-like API, without having to write any extra code. Once the model server starts (vLLM will output “Application startup complete.”), we can send requests to this LLM, as shown below:

```
$ curl http://localhost:8000/v1/completions -H "Content-Type: application/json" -d '{ "model": "facebook/opt-125m", }
```

```

        "prompt": "This summer we will"
    }

{
    "id": "cmpl-15427a0bd94a4acd834f7fdad5d3a0f2",
    "object": "text_completion",
    "created": 1744061002,
    "model": "facebook/opt-125m",
    "choices": [{"index": 0, "text": "# be a big step closer to\nseeing truly meaningful galaxies in skies.\nI'm", "logprobs": null, "finish_reason": "length", "stop_reason": null, "prompt_logprobs": null}, {"usage": {"prompt_tokens": 6, "total_tokens": 22, "completion_tokens": 16, "prompt_tokens_details": null}}]
}

```

The vLLM CLI is ideal for testing or running a production-style service locally or inside a container. However, when you need to run inference jobs as standalone tasks, batch processes, or integrate inference logic more deeply into your application code, the Python API is a better fit. Rather than spinning up an HTTP server, you instantiate the vLLM engine directly.

Listing 8.1 Creating a vLLM model server

```

from vllm import LLM

prompt = "What are we having for dinner?" #A

sampling_params = SamplingParams(temperature=0.8, top_p=0.95)

llm = LLM(model="facebook/opt-125m") #B

outputs = llm.generate(prompt, sampling_params) #C

print(outputs[0].outputs[0].text) #D

#Outputs "Uh, I'll be having chicken stew, mushrooms, and what have you"

#A Defines the input prompt for the model.
#B Initializes vLLM's engine and the OPT-125M model for offline inference
#C Sends the prompt and sampling settings to the LLM instance for inference.
#D Extracts and prints the generated text from the output response.

```

Whether you choose online or offline mode, both methods leverage the same underlying vLLM engine. The CLI approach helps you quickly spin up an OpenAI-compatible service, while the Python interface offers more flexibility and control. Now let's see how to integrate these approaches into Kubernetes for larger-scale deployments.

8.3.1 Creating an LLM service on Kubernetes

Let's deploy a vLLM service on a Kubernetes cluster using the official Helm chart from vLLM's production stack. We'll deploy the same lightweight text generation

model ("facebook/opt-125m") as the previous section. The key difference being that the model will now run inside a Kubernetes cluster.

Follow these steps to create the service in your cluster:

```
$ cd Book/Chapter 8/vllm
$ helm repo add vllm https://vllm-project.github.io/production-
stack
$ helm install vllm vllm/vllm-stack -f vllm-example.yaml
```

WARNING. This deployment needs a node with an NVIDIA GPU to run properly. If you're using the Amazon EKS cluster from this book's code repository, keep in mind that EKS will automatically spin up GPU-enabled nodes to fulfill this requirement, incurring additional costs in your AWS bill.

Wait until Pods are in "Running" status:

```
$ kubectl get pods -w
```

Once these Pods are running, you can list the Services that the chart created:

```
$ kubectl get svc
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP
vllm-engine-   ClusterIP  172.20.96.208  <none>        80/TCP
service
vllm-router-  ClusterIP  172.20.181.240 <none>        80/TCP
service
```

To access the router from your local machine, use kubectl port-forward. Suppose your router Service is named vllm-router-service:

```
$ kubectl port-forward svc/vllm-router-service 8000:80
```

Now the router's API is reachable at localhost:8000, letting you interact with the OpenAI-compatible endpoints. For example, list available models:

```
$ curl http://localhost:8000/v1/models
{"object": "list", "data": [{"id": "facebook/opt-125m", "object": "model", "created": 1744146360, "owned_by": "vllm", "root": null}]}%
```

Or invoke the model:

```
$ curl http://localhost:8000/v1/completions \
-H "Content-Type: application/json" \
-d '{
```

```

        "model": "facebook/opt-125m",
        "prompt": "What are we having for dinner?",
        "max_tokens": 16
    }
    {"id":"cmpl-
eec25e73232546028d4c39e31204602a","object":"text_completion","cr-
eated":1744146585,"model":"facebook/opt-
125m","choices":[{"index":0,"text":" Tough questions, but
neither have I been able to answer.
^we're","logprobs":null,"finish_reason":"length","stop_reason":n-
ull,"prompt_logprobs":null}],"usage":{"prompt_tokens":8,"total_t-
okens":24,"completion_tokens":16,"prompt_tokens_details":null}}

```

To delete the LLM server, run:

```
$ helm uninstall vllm
```

This will delete the router and engine Pods along with their dependencies.

8.3.2 *Understanding the deployment*

Now that we've deployed a vLLM service using Helm, let's take a closer look at the YAML file used to configure the vLLM engine. Listing 8.2 contains the Helm values file (vllm-example.yaml). It specifies which model to load and how many resources (CPU, RAM, and GPU) to request.

Listing 8.2 vLLM Helm values file defining the model serving engine

```

servingEngineSpec:
  modelSpec:
    - name: "opt125m" #A
      repository: "vllm/vllm-openai" #B
      tag: "v0.8.2"
      modelURL: "facebook/opt-125m" #C
      replicaCount: 1 #D
      requestCPU: 6 #E
      requestMemory: "16Gi" #F
      requestGPU: 1 #G
#A Name of the deployment (usually the name of the model being deployed)
#B The container image repository and tag. The vLLM project
provides prebuilt images here. vLLM will use this image to create the
engine Pods.
#C The actual model identifier on Hugging Face or another provider, such as
"facebook/opt-125m." This tells vLLM which model weights to download and load at
startup.
#D Number of engine replicas to start with. This number can be later changed to
scale the deployment.
#E The amount of CPU cores to reserve for each Pod.
#F The amount of memory to reserve for each Pod.
#G Number of GPUs to be assigned to each engine Pod.

```

This deployment creates the following Kubernetes resources:

19. Router Deployment – Each router Pod exposes an OpenAI-compatible API endpoint, receives requests from external clients, and forwards them to the appropriate engine Pod. Because the router is stateless, you can scale this Deployment up or down based on traffic volume without worrying about losing any session data.
20. Engine Deployment - Each engine Pod loads one or more models, handles GPU-based inference, and manages the KV cache for its assigned sessions. Scaling the engine deployment typically revolves around how many GPUs you have and how much throughput you need for large language model serving.
21. Router Service – This is the primary entry point for your LLM service, where you might attach a LoadBalancer or configure Ingress rules to route external requests. Since the router service is responsible for distributing traffic to router Pods, you'll typically reference it when performing a port-forward or exposing it externally for client access.
22. Engine Service – End users won't typically connect to the engine service directly, as it's primarily there so the router can discover and route requests to the engines. This separation ensures that the router can handle authentication, request batching, and error handling, leaving the engine Pods to focus solely on inference tasks.

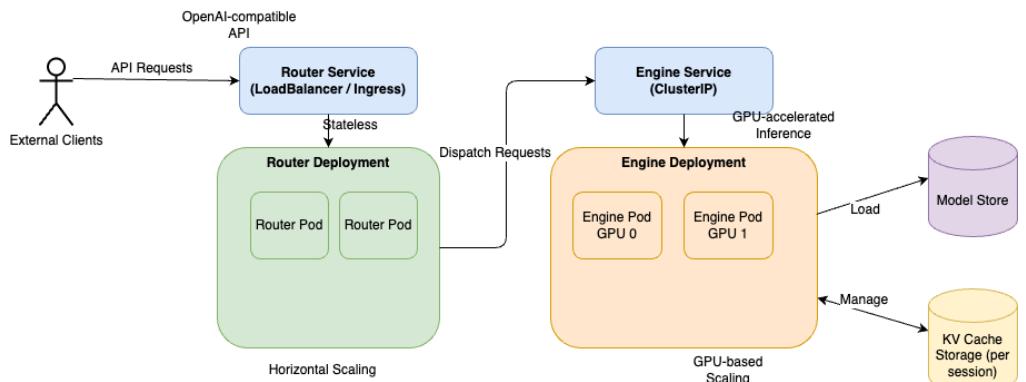


Figure 8.2 A two-tier deployment with stateless router pods managing API requests and engine pods handling GPU-accelerated model inference.

Because the router and engine deployments run in separate sets of Pods, you can scale them independently based on different resource requirements. If incoming traffic spikes and the router Pods become overloaded, you can increase the router replica count without adding GPUs. On the other hand, if the inference load itself becomes the bottleneck (for instance, if GPU utilization is maxed out), you can scale the engine deployment upward, adding more GPU-allocated Pods as needed. This flexibility helps you tune your vLLM setup so you're only paying for the compute power you actually need, whether it's more front-end capacity to handle bursts of requests or more back-end GPU power to handle complex model operations.

8.3.3 *Hosting multiple models*

vLLM's distributed architecture also allows multiple inference engines, each serving a different model, to be managed by a single router. In listing 8.3, we define two separate models, "DeepScaleR-1.5B-Preview" and "opt-125m," and assign a single GPU to each. DeepScaleR-1.5B-Preview is a larger model designed for high-quality summarization and reasoning tasks, while opt-125m is a lightweight model well-suited for low-latency completions or resource-constrained environments.

When you deploy this configuration, the router automatically routes requests to the correct engine based on which model is specified in the incoming API call. Each model can be scaled independently, providing flexibility to allocate more GPUs for larger models or more replicas to handle heavier traffic. This multi-model arrangement simplifies your infrastructure by consolidating inference behind a single endpoint, while still letting you tailor resource usage for each individual engine.

Listing 8.3 Serving multiple models with vLLM (vllm-multiple-models.yaml)

```
servingEngineSpec:  
  modelSpec:  
    - name: "deepscale" #  
      repository: "vllm/vllm-openai"  
      tag: "latest"  
      modelURL: "agentica-org/DeepScaleR-1.5B-Preview"  
      replicaCount: 1  
      requestCPU: 6  
      requestMemory: "16Gi"  
      requestGPU: 1  
    - name: "opt125m"  
      repository: "vllm/vllm-openai"  
      tag: "latest"  
      modelURL: "facebook/opt-125m"  
      replicaCount: 1
```

```

requestCPU: 6
requestMemory: "16Gi"
requestGPU: 1

```

When you deploy this configuration (by running `helm install vllm/vllm-stack -f vllm-multiple-models.yaml`), vLLM creates Kubernetes Deployments for each model engine. This allows you to fine tune resource requests based on the needs of each model. For example, an ultra-large LLM might require multiple GPUs per replica, whereas a smaller or specialized model could run on a single GPU with a larger number of replicas to handle traffic patterns with sudden spikes in request volume. Each model deployment can be scaled independently, giving you the flexibility to allocate compute resources based on model size, usage patterns, or latency requirements. The vLLM router ensures that calls to each model remain isolated at the engine layer, preventing resource conflicts and keeping inference latency predictable.

In a production environment, this flexibility lets you expand your LLM offerings without adding unnecessary complexity to your architecture. You can introduce new models by simply appending another entry to your Helm values, then scale each engine to meet real-world demand. Whether you need two models or twenty, the combination of independent engine deployments and a centralized router creates a unified, cost-effective solution for multi-model inference on Kubernetes.

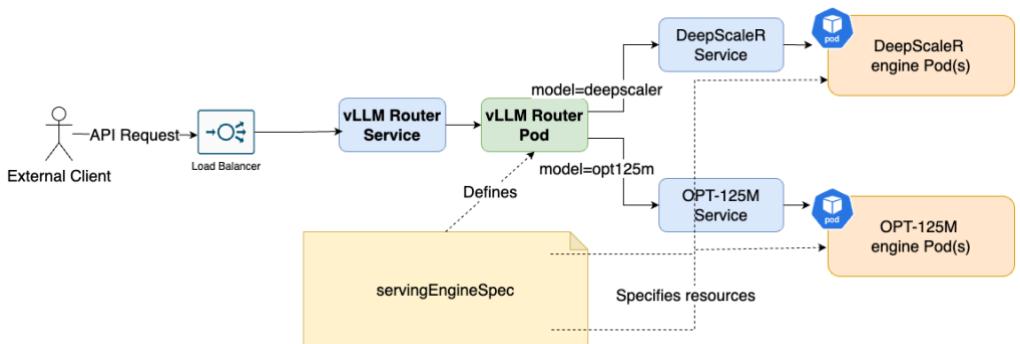


Figure 8.3 Multi-model serving architecture on Kubernetes. External API requests are routed through a centralized Router Service and Router Pod, which dispatch requests to specific model services (e.g., DeepScaleR or OPT-125M) based on routing logic defined in `servingEngineSpec`. Each model is deployed as a dedicated engine Pod with its own resource specifications (CPU, memory, GPU) and exposed via a corresponding Kubernetes Service.

This configuration also streamlines upgrades and rollbacks. Since each engine is independent, updating a single model involves changing only that model's entry in your Helm values file. Kubernetes then handles the rolling updates or deployments without affecting the availability of other engines or the router.

This architecture doesn't just simplify infrastructure, it opens up a wide array of design possibilities for software development teams. By hosting multiple models behind the same endpoint, applications can expose specialized Generative AI (GenAI) capabilities under distinct API routes. For example, you might deploy one LLM that handles translation from German to English, and another that performs text summarization. Client applications can interact with the same base URL, calling `/translate` or `/summarize` to invoke the appropriate model. From the developer's point of view, it's a unified API. Behind the scenes, the vLLM router intelligently dispatches each request to the correct engine based on the model name specified in the request payload.

This setup allows teams to build modular, composable AI services by offering:

- Isolated iteration - Update or retrain one model without disrupting others.
- Model-specific resource limits - Allocate GPUs, memory, or replicas based on each model's needs.
- Clean routing logic - Route requests by model name, mapping naturally to application-level concerns.
- Simplified client integration - Expose a single endpoint, no need for clients to juggle multiple URLs or credentials.

Everything flows through a single, centralized gateway, while the underlying models operate independently with full isolation and scalability. This model-aware routing lets you scale your AI platform like a microservice architecture, while still keeping operations efficient and manageable.

This configuration also streamlines upgrades and rollbacks. Since each engine is independent, updating a single model involves changing only that model's entry in your Helm values file. Kubernetes then handles the rolling updates or deployments without affecting the availability of other engines or the router.

This architecture also creates an opportunity for monetization. By pairing vLLM with an API gateway, you can expose multiple LLM-powered services, like translation or summarization, as metered API endpoints. Requests can be rate-limited, authenticated, and billed per usage, enabling teams to offer GenAI as a service. An open source, Kubernetes-native option like Kong Gateway (<https://github.com/Kong/kong>) integrates seamlessly with vLLM deployments.

Kong supports authentication plugins, request logging, usage tracking, and billing hooks, making it a strong choice for productizing multi-model inference workloads. With the right setup, you can operate a self-hosted AI API platform that functions similarly to commercial offerings, but gives you full control over infrastructure, models, and pricing.

8.3.4 Serving model variants with Multi-LoRA

In many real-world applications, you'll need to serve multiple fine-tuned variants of the same base model. For example, imagine building chatbots to support different product lines within your company, each product may require its own LLM fine-tuned on relevant support documentation. Rather than hosting a separate full model for each variant, which would be both memory- and compute-intensive, you can use a technique called *LoRA (Low-Rank Adaptation)*.

vLLM supports multi-LoRA, allowing you to load a base model once and serve multiple LoRA adapters side by side from the same engine Pod. *Multi-LoRA* is a technique that enables dynamic switching between different Low-Rank Adaptation modules at inference time without needing to reload the entire foundation model. This approach avoids duplicating memory usage and significantly improves GPU efficiency since only the base model weights remain resident in GPU memory while the much smaller adapter weights can be swapped on demand. When a request comes in, you simply specify which adapter to use via the API, and vLLM dynamically applies the corresponding weights during inference. For example, you might have one adapter fine-tuned for customer service, another for technical writing, and a third for creative content, all served from the same model instance. This is ideal for scenarios where multiple teams or products share a common foundation model but require task-specific behavior. With multi-LoRA, vLLM gives you the flexibility to support personalization at scale without needing to spin up separate infrastructure for every variant, dramatically reducing both operational costs and inference latency compared to deploying separate model instances.

What is LoRA?

LoRA (Low-Rank Adaptation) is a parameter-efficient fine-tuning technique that adds small trainable "adapter" modules to a frozen pre-trained model. Instead of updating all model weights during fine-tuning, which would require storing a complete copy of the model for each variant, LoRA inserts compact rank decomposition matrices that modify the behavior of the original model.

These adapters typically add only 1-4% to the size of the original model while enabling significant customization of the model's capabilities.

For example, with a base model like llama-2-7b-hf (which requires approximately 28GB of GPU memory), traditional fine-tuning would require separate 28GB allocations for each specialized version. In contrast, LoRA adapters for the same model might only add 100-400MB each, allowing you to host multiple specialized versions with minimal additional memory overhead.

LoRA adapter support in vLLM depends on the architecture of the base model. Not all models support LoRA integration out of the box. If you're unsure whether a given model supports LoRA, check the Hugging Face model card or test the adapter loading process during development.

Currently, vLLM does not automatically download LoRA adapters from Hugging Face. If you're deploying a model with LoRA support, you'll need to manually download the adapter files into your Pod or mount them from a shared volume (as explained in vLLM Production Stack documentation: https://docs.vllm.ai/projects/production-stack/en/latest/tutorials/lora_load.html). This step is required before the engine can load the adapter at runtime.

For production use, the best practice is to download or mount the LoRA adapters before the engine starts, using a persistent volume, an initContainer, or pre-baking the adapters into the container image. This ensures that the engine can discover and load all adapters at boot time. However, do keep in mind that you can update LoRA config without requiring a restart of the Pod or the underlying engine process.

Looking ahead, the vLLM team is planning to streamline this workflow even further with native support for downloading LoRA adapters directly from Hugging Face via Helm configuration. This upcoming enhancement will reduce the operational overhead of managing adapters across distributed environments.

NOTE Please see vLLM documentation to know more about vLLM's LoRA support. To learn more about the future of multi-LoRA management in Kubernetes visit <https://github.com/vllm-project/production-stack/issues/300>.

8.3.5 Performing Updates without Downtime

Anytime you want to change or upgrade the LLM model in a Kubernetes-based vLLM deployment, it's crucial to do it in a way that doesn't disrupt service for your users. Kubernetes has built-in support for rolling updates, which allow you to replace Pods gradually without causing downtime.

To perform a rolling update, first modify your Helm values.yaml file with the new model configuration (e.g., changing the modelURL to point to a newer or different model). After saving your changes, you'll deploy the update like this:

```
$ helm upgrade vllm vllm/vllm-stack -f updated-values.yaml
```

When you do this, Kubernetes initiates a rolling deployment by creating new engine Pods based on the updated configuration, while keeping your existing Pods running and serving traffic.

Kubernetes monitors the new Pods, using health checks (readiness probes) to ensure they're fully operational and capable of serving requests. Only after a new Pod passes these health checks will Kubernetes start routing traffic to it. This helps ensure that only healthy Pods handle live user requests.

The reliability of rolling updates hinges on the accuracy of readiness probes. These probes signal to Kubernetes when a Pod is ready to receive traffic. For vLLM, a readiness probe might verify that the server's API endpoint is responding correctly. An example of a readiness probe configuration within the values.yaml file could look like this:

```
readinessProbe:  
  httpGet:  
    path: /health # or the specific vLLM health endpoint  
    port: 8080 # or the vLLM service port  
  initialDelaySeconds: 15 # Wait 15 seconds after Pod start  
  periodSeconds: 10 # Check every 10 seconds  
  timeoutSeconds: 1 # Timeout after 1 seconds
```

It is important to set initialDelaySeconds to a value that allows the vLLM server sufficient time to load the model. Monitor your Pod startup times to determine an appropriate value, you can observe how long model loading takes in your environment by checking container logs during startup phases and adding a small buffer. The periodSeconds and timeoutSeconds values should be adjusted based on the expected response time of the vLLM server. Monitor your Pod startup times to determine an appropriate value.

Throughout this process, the router only forwards traffic to Pods that are ready to serve traffic. As new Pods become ready, the router starts forwarding them requests. Conversely, when older Pods terminate, Kubernetes stops

sending them traffic by removing them from the engine Service's list of Endpoints (this list defines which Pods should receive traffic). When Pods in the new Deployment pass readiness probes, Kubernetes gracefully terminates the older Pods. Any ongoing inference tasks are completed before termination, ensuring a smooth transition with minimal user disruption.

To further ensure zero downtime during rolling updates, you can configure a PodDisruptionBudget (PDB) (<https://kubernetes.io/docs/concepts/workloads/pods/disruptions/>) for the engine deployment. A PDB limits the number of Pods that can be voluntarily evicted during maintenance events such as node upgrades or scaling actions. For example, you might define a PDB that ensures at least one engine Pod is always available, preventing situations where all engine Pods are restarted at once and temporarily take your model offline.

8.3.6 Optimizing Model Loading and Startup

So far, every vLLM deployment we've created downloads models from Hugging Face Hub each time an engine Pod starts. This process can take several minutes, significantly delaying the time before a new engine Pod is ready to serve traffic. Meanwhile, we're still paying for expensive GPUs.

To reduce startup time, especially in multi-replica scenarios, vLLM supports model caching via Persistent Volumes (PV). Rather than having each engine Pod download the model individually, all engine Pods can be configured to mount the same Persistent Volume Claim (PVC), which stores the model artifacts. On first use, the model is downloaded to the shared volume. Subsequent Pods then reuse those cached files.

This only works if the backing storage supports simultaneous attachment to multiple Pods. That means the volume must be provisioned using a storage class that supports ReadWriteMany (RWX) access mode. Examples include:

- Amazon EFS
- Google Filestore
- Azure Files
- NFS-backed volumes for on-prem or hybrid clusters

In contrast, volumes with ReadWriteOnce access (such as AWS EBS or Azure Disks) cannot be mounted by more than one Pod at a time, which makes them unsuitable for this type of model caching. Using ReadWriteOnce volumes will cause issues in multi-replica vLLM deployments. Each pod will try to attach the same volume, which is not allowed, and will prevent the pods from starting correctly.

Listing 8.4 contains the definition of a vLLM deployment in which we add a 50Gi PV. When we deploy this Helm chart, Kubernetes creates a Persistent Volume Claim and adds it to the engine Pod.

Listing 8.4 Adding PV to vLLM

```
servingEngineSpec:  
  modelSpec:  
    - name: "opt125m"  
      repository: "vllm/vllm-openai"  
      tag: "latest"  
      modelURL: "facebook/opt-125m"  
      replicaCount: 1  
      requestCPU: 6  
      requestMemory: "16Gi"  
      requestGPU: 1  
      pvcStorage: "50Gi" #A  
      #A Adds a 50 GB PV to the engine Pods
```

However, keep in mind that loading large models from cloud-based shared storage solutions like Amazon EFS might not always be faster than downloading them directly from Hugging Face Hub, particularly if network conditions are good. For optimal startup performance and efficient scaling, consider using high-performance storage options (they are also costlier than NFS-based solutions) such as Amazon FSx for Lustre (AWS), Azure NetApp Files or Azure Files Premium (Azure), or Google Parallelstore (GCP). These provide much higher throughput and lower latency, significantly outperforming standard shared file storage options.

If your cluster nodes include local SSDs, such as NVMe drives commonly available on many cloud GPU virtual machines, you can further improve performance by configuring PV to use these local drives. Local SSDs provide extremely low latency and high IOPS, ensuring the fastest possible model loading times. However, be aware that local storage is ephemeral and data could be lost if the node fails or restarts, so consider this carefully for critical workloads.

8.3.7 Loading models from object storage

An alternative to Persistent Volumes is storing model weights in a cloud object storage service such as Amazon S3, Google Cloud Storage, or Azure Blob Storage. This approach works well when your model isn't stored in Hugging Face Hub or you'd like to avoid downloading it from the internet every time an engine Pod starts.

If you choose to store model weights in Amazon S3 (or similar object storage services like GCS or Azure Blob), one common pattern is to use an Init Container

to download the model files before the main vLLM engine container starts. You can define an Init Container in your vLLM Helm values file.

When a new engine Pod starts, Kubernetes runs the Init Container before starting the engine container within the Pod. The Init Container pulls the model from S3 and places it in a shared volume that the main container can read from.

One thing to keep in mind is that for large models, downloading the data from the object storage can still add significant cold-start latency, particularly under burst scaling scenarios. This method is best suited when shared PVs are unavailable or the underlying shared storage systems is a bottleneck or when you want maximum flexibility in managing and versioning models using your object store.

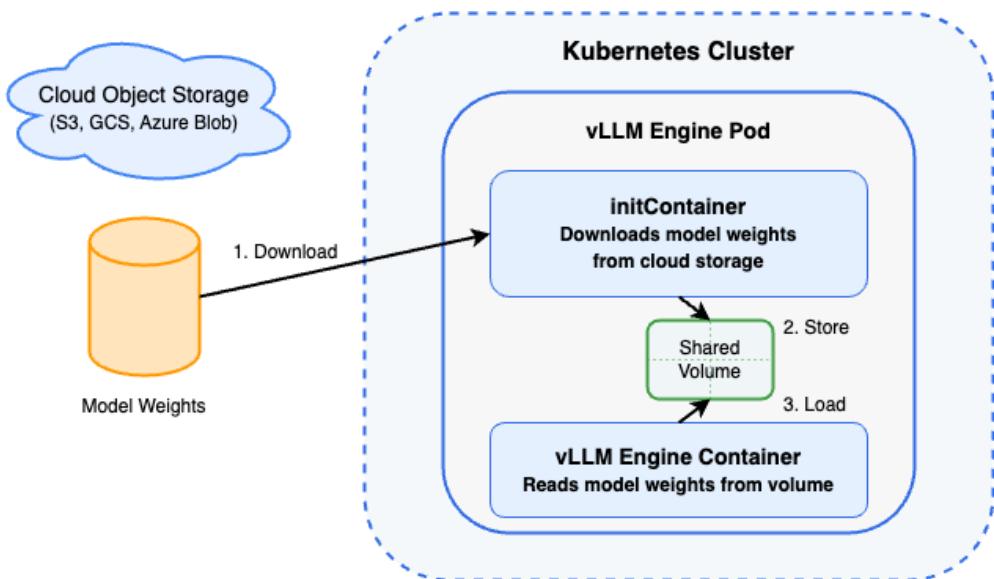


Figure 8.4 Downloading weights from object storage services. Note that initContainer downloads model before the vLLM engine container starts

8.3.8 Using MinIO as a Model Cache

Another way of caching model weights is by using MinIO (<https://github.com/minio/minio>) as a caching layer. *MinIO* is an open-source, S3-compatible object storage system that you can deploy in your Kubernetes cluster. It provides S3 API compatibility, making it interoperable with any S3-

designed application, while featuring a Kubernetes-native design for seamless container orchestration integration.

In this architecture, when a new vLLM engine pod starts, it first checks the MinIO cache to see if the requested model is already available in the cluster. If the model exists in MinIO, it's served directly to the vLLM pod with very low latency, avoiding external downloads. If the model isn't found in MinIO, it's downloaded from the upstream source (like Hugging Face Hub), stored in the MinIO cache, and then served to the vLLM engine pod. This pattern is especially valuable for large language models where downloading from external sources can take several minutes and consume significant bandwidth.

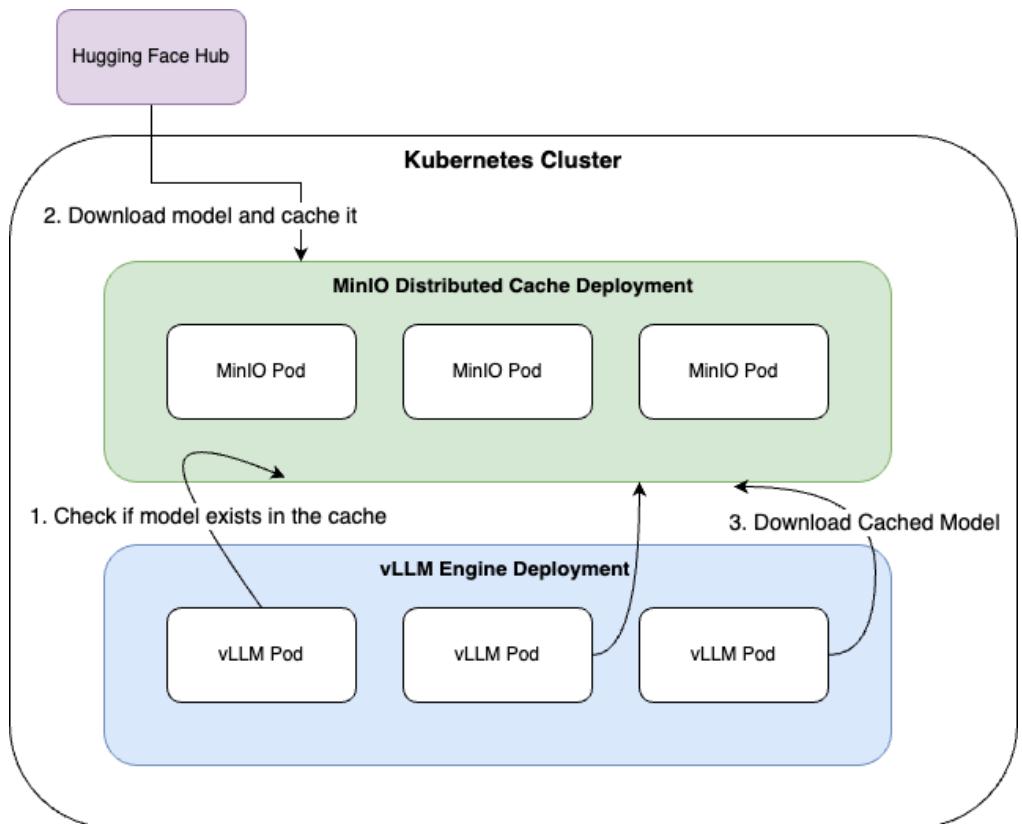


Figure 8.5 An initContainer can be used to download model weights from MinIO running inside the cluster. This in-cluster caching layer dramatically reduces the time vLLM takes to start up and scale.

The initial cold-start latency when a model isn't yet cached is a consideration, as the first pod requesting a new model will still experience the full download time. However, this is a one-time cost that benefits all future pod startups.

8.3.9 Reducing cold startup with image caching

Another key factor affecting startup latency in vLLM deployments is container image pull time. The vLLM engine image is large, typically around 8 GB, and depending on your cluster setup, downloading this image can take several minutes. This delay can significantly impact autoscaling responsiveness, Pod restarts, and rolling updates.

To address this, there are several techniques Kubernetes operators can use to cache container images and reduce cold-start latency.

INCLUDING CONTAINER IMAGES IN THE WORKER NODE'S IMAGE

If you're using custom images for your Kubernetes nodes (e.g., with Amazon EKS or self-managed clusters), you can bake the vLLM image into your node AMI. When new GPU nodes are created, they already contain the vLLM engine image on disk. This eliminates the need to pull the image from a remote registry, which is especially beneficial in high-scale or bursty environments.

Some managed Kubernetes services also provide native support for this. For example, Google Kubernetes Engine (GKE) offers a feature called Secondary Boot Disk (SBD) that stores container images on a second disk mounted to each node. When a Pod starts, it can access the container image directly from the attached disk, bypassing the need to pull it from a registry. This reduces cold start latency substantially. The image must be preloaded into the disk image before deployment, typically via automation.

PRE-PULLING WITH KUBERNETES IMAGE PULLER

[Kubernetes Image Puller](https://github.com/GoogleContainerTools/k8s-imager) (<https://github.com/GoogleContainerTools/k8s-imager>) is an open source project that simplifies preloading images across all nodes in your cluster. It automatically creates DaemonSets for each specified image and ensures the image is cached on every node. It's Kubernetes-native, version-aware, and can optionally remove the images after use. This procedure ensures that when vLLM engine Pods are scheduled, the container image is already present on the node, eliminating startup delays caused by pulling large images.

ENABLING LAZY IMAGE LOADING

Container runtimes like containerd (default runtime in many Kubernetes clusters) support lazy image loading, where images are streamed in on-demand rather than being fully pulled before start. This can significantly speed up startup time for large images, especially when only a portion of the image layers are used during initialization.

Two popular formats for lazy loading are:

- estargz (extended stargz) (<https://github.com/containerd/stargz-snapshotter>): A container image format supported by containerd with the Stargz Snapshotter. It reorders content within image layers for better streaming performance and enables fetching individual files from a remote registry without pulling the entire image.
- SOCI (Seekable OCI)(<https://github.com/awslabs/soci-snapshotter>): An open source lazy image technology developed by AWS. SOCI builds a seekable index for OCI images, allowing images to be mounted and started before the full download. However, as of writing, SOCI is still under active development and not considered production-ready for use Kubernetes. It's suitable for testing or controlled environments, but not yet ideal for critical workloads.

8.4 Autoscaling vLLM

In the previous sections, you learned that vLLM engine Pods are managed as part of a Kubernetes Deployment. You can control the number of replicas directly in your Helm values file during installation, or later by manually updating the Deployment. This approach works fine for static or experimental workloads, but in production environments where load fluctuates, scaling should be automated.

That's where Kubernetes Horizontal Pod Autoscaling (HPA) comes in. HPA allows you to automatically adjust the number of replicas based on observed load. But here's the challenge: LLM inference workloads don't behave like traditional web applications. Kubernetes HPA was originally designed to react to CPU and memory usage, which may not be the best signals for GPU-bound workloads like vLLM.

Kubernetes HPA supports multiple metric sources to drive scaling decisions. The most common is the Metrics Server (<https://github.com/kubernetes-sigs/metrics-server>), which provides built-in support for CPU and memory usage metrics by aggregating data from all nodes and Pods in the cluster. For more advanced use cases, HPA also supports custom metrics through the `custom.metrics.k8s.io` API, which can be populated by Prometheus using the Prometheus Adapter.

NOTE. If you aren't familiar with Prometheus, some of these terms may be new to you. We will go over observability in much more details later in the book.

Custom metrics are application-specific and offer better visibility into actual workload behavior. Additionally, Kubernetes supports external metrics through the `external.metrics.k8s.io` API, allowing you to scale based on signals from systems outside the cluster, such as cloud queues, APIs, or monitoring platforms.

When scaling on CPU or memory, HPA periodically queries the Metrics Server for resource usage data across all Pods in the target Deployment. It then computes the average CPU or memory utilization across all Pods and compares that against a target threshold defined in the HPA spec. For example, let's say you have a Deployment with two Pods and you configure HPA to scale the Deployment if average CPU usage across all its Pods exceeds 80%. Should the aggregate CPU usage go beyond this threshold, HPA will increase the number of replica Pods in the Deployment.

This approach works well for CPU-bound applications, but with GPU workloads like LLM inference, CPU usage often remains low even when the GPU is saturated. That's why autoscaling vLLM using custom metrics is more effective and provides a better signal of the model server(s) are performing.

You can see this in action in figure 8.5 where on the left side, we see two vLLM engine pods with low CPU utilization (30% and 25%) but nearly saturated GPUs (95% and 98%). The HPA decision box shows that with a target CPU of 80% and actual average of just 27.5%, no scaling occurs despite the GPU bottleneck. In contrast, the right side demonstrates custom metrics-based scaling using the same pods but incorporating queue depth metrics (45 and 39). Here, the HPA decision box reveals that with a target queue depth of 25 and an actual average of 42, the system correctly triggers scaling up. This comparison effectively highlights how custom metrics provide much more relevant signals for autoscaling GPU-dependent workloads like vLLM, where CPU utilization remains deceptively low even when the actual processing capacity is overwhelmed.

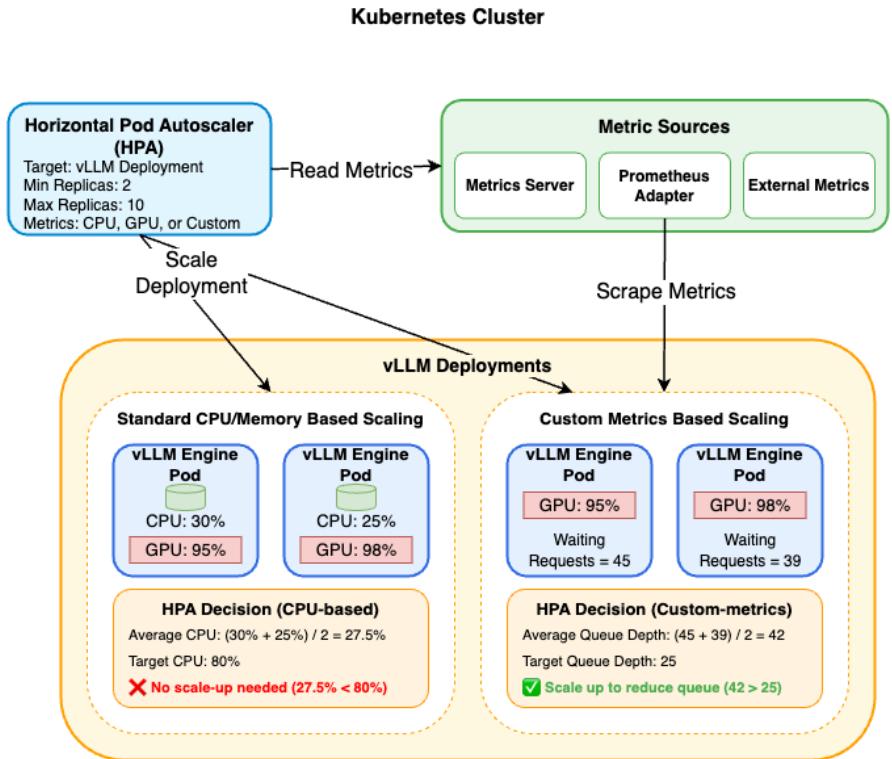


Figure 8.6 Kubernetes Horizontal Pod Autoscaling (HPA) for vLLM Workloads. For GPU workloads like vLLM, CPU metrics may remain low while GPUs are saturated, making custom metrics more effective for scaling.

vLLM exposes a number of metrics (<https://docs.vllm.ai/en/latest/serving/metrics.html>) that can be used to monitor the health of the system. The metric `vllm:num_requests_waiting` is a Prometheus gauge exposed by vLLM to indicate the number of inference requests currently queued and awaiting processing by the engine. This metric indicates vLLM engine's load and is usually used for autoscaling vLLM engine Pods.

In essence, as your vLLM-powered LLM service gains more users the number of inference requests begins to outpace the available processing capacity. Note that these “users” could be external customers or internal teams like HR staff querying documents or developers using an LLM-powered coding assistant. As more users send requests, the queue of pending requests grows, driving up the

`vllm:num_requests_waiting` metric. At this point, Kubernetes needs to spin up additional vLLM engine Pods (and by extension, more GPU resources) to maintain performance and keep latency low.

NOTE The vLLM documentation includes an excellent, step-by-step walkthrough tutorial for setting up Horizontal Pod Autoscaling using Prometheus metrics (<https://github.com/vllm-project/production-stack/blob/45482277cf4f827e19d36c95c3a4b863ce28cb25/tutorials/10-horizontal-autoscaling.md>). Rather than repeating the same instructions here, we encourage readers to follow the official exercise on their own to better understand how it works in practice. Later in this book, we'll walk through deploying Prometheus in our Kubernetes cluster and show vLLM autoscaling in action, so you'll get hands-on experience seeing how `vllm:num_requests_waiting` can drive real-time scaling of inference Pods.

8.4.1 Benchmarking a vLLM Deployment

Before rolling out your vLLM service in production, it's critical to benchmark your setup. This helps you understand the performance limits of your deployment and ensures that you're not overspending on resources, or worse, under-provisioned when traffic hits.

Here are the core dimensions to test:

- Throughput – Measure how many tokens per second your setup can handle under different batch sizes and input lengths.
- Latency – Capture both time-to-first-token and average time per generated token, especially under concurrent load.
- Concurrency – Identify the tipping point where increasing simultaneous requests starts to degrade performance or queueing grows.
- Resource Utilization – Monitor GPU memory usage, GPU utilization %, and CPU saturation to catch bottlenecks.

You can use tools like locust (<https://locust.io/>) or hey (<https://github.com/rakyll/hey>) to simulate realistic traffic. For example:

```
$ hey -n 100 -c 10 -m POST -H "Content-Type: application/json" \
-d '{"model":"your-model","prompt":"What is
MLOps?","max_tokens":100}' \
http://your-vllm-service/v1/completions
```

While benchmarking, monitor resource utilization closely. GPU memory usage shows how efficiently the KV cache is managed and whether you're approaching memory limits. GPU compute utilization (typically visible via `nvidia-smi`) reveals if your workload is fully utilizing available compute capacity. CPU usage may become a bottleneck in preprocessing or postprocessing steps, particularly with high request volumes.

Run these tests across various configurations while changing batch size, GPU type, engine replicas and compare results. Sometimes this is the most practical way to identify the sweet spot between cost and performance for a given workload.

8.5 *Distributed inference with vLLM*

Within a few years, we've seen models go from billions to hundreds of billions of parameters. As LLMs grow in size, they quickly exceed the memory capacity of a single GPU. Distributed inference with vLLM solves this problem by splitting model execution across multiple GPUs, and even across nodes. This allows massive models to be served efficiently, even in environments where no single GPU has enough VRAM to load the full model.

At a technical level, vLLM uses tensor parallelism to partition model weights and distribute execution. This is typically configured by assigning multiple GPUs to a single Pod (for example, `nvidia.com/gpu=4`) The distributed execution engine is tightly integrated with Kubernetes networking and GPU scheduling, so you get high throughput with minimal tuning.

In vLLM, distributed inference scales through two methods:

- *Tensor parallelism* splits computation across GPUs within a single node. For models that fit within a single node's combined GPU memory, tensor parallelism shines. It splits matrices, attention heads, and other computations across devices, with each GPU handling a portion of the calculation before results are synchronized. This approach minimizes communication overhead since the GPUs typically share fast NVLink or PCIe connections.
- *Pipeline parallelism* distributes model layers across multiple nodes. When models grow beyond what even a multi-GPU node can handle, pipeline parallelism comes into play. This technique divides the model vertically across its layers, assigning different stages to separate nodes. As requests flow through the system, each node processes its assigned layers before passing intermediate activations to the next node. This enables even 100B+ parameter models to run efficiently across standard GPU clusters.

In many situations, we may employ both of these parallelism approaches simultaneously.

8.5.1 *Multi node inference*

For models that exceed even the multi-GPU capacity of a single node, vLLM supports multi-node distributed inference through its integration with Ray. Both vLLM and Ray are developed by the same team at UC Berkeley’s SkyLab, which means the integration between the two is tightly aligned and actively maintained. Ray allows vLLM to scale inference horizontally across GPU nodes in a Kubernetes cluster, unlocking model sizes and workloads that would otherwise be impossible to serve.

When you run vLLM with Ray, you gain reliability and orchestration features that are otherwise manual or missing in a vanilla vLLM deployment:

- Scalability – Ray allows you to scale vLLM horizontally by adding more nodes. You can run more engine replicas across a cluster to handle higher request volumes, without modifying the app or traffic layer. Without Ray, you’d have to manage this scaling yourself, either via Kubernetes Deployments or external tooling.
- Load balancing – Ray automatically distributes inference requests across available replicas. It handles routing under the hood, ensuring even distribution and avoiding hot spots. Without Ray, you’d need to implement your own load balancing logic, usually by customizing the vLLM router or inserting a service mesh.
- Fault tolerance – If Ray detects a failed worker, it can immediately reschedule pending tasks to other healthy workers without waiting for Kubernetes to restart the Pod.
- Orchestration – Ray manages the entire lifecycle of your vLLM replicas. It can deploy, autoscale, and remove workers based on current workload. Without Ray, orchestration means building a patchwork of Helm charts, HPAs, and metrics adapters.

We’ve included a complete Ray + vLLM model serving script in the book’s GitHub repository under the name `vllm-ray-serve.py`. Listing 8.5 shows the snippet of a manifest that lets you deploy a vLLM engine backed by Ray Serve, complete with an OpenAI-compatible API, autoscaling logic, and support for distributed inference. We’ve annotated the script, so we won’t go through it line by line in this chapter. Doing so would require a deep dive into the internal APIs of vLLM, Ray Serve, and FastAPI, topics that go beyond the scope of this book. We encourage readers interested in the implementation details to explore the

script and official documentation (https://docs.vllm.ai/en/latest/serving/distributed_serving.html).

Listing 8.5 KubeRay manifest for serving an LLM using vLLM on Ray Serve (ray-service.vllm.yaml)

```
apiVersion: ray.io/v1
kind: RayService
metadata:
  name: mistral-7b
spec:
  serveConfigV2: |#A
    applications:
    - name: llm #B
      route_prefix: / #C
      import_path: ray-operator.config.samples.vllm.serve:Model
  #D
  deployments:
  - name: VLLMDeployment
    num_replicas: 1 #E
    ray_actor_options:
      num_cpus: 8
    runtime_env:
      working_dir: "https://github.com/ray-project/kuberay/archive/master.zip" #F
      pip: ["vllm==0.6.1.post2"] #G
    env_vars:
      MODEL_ID: "mistralai/Mistral-7B-Instruct-v0.2" #H
      TENSOR_PARALLELISM: "2"
      PIPELINE_PARALLELISM: "1"
  rayClusterConfig: #I
    headGroupSpec:
      ...
    workerGroupSpecs:
      ...
#A Ray Serve Configuration
#B Application name
#C Routes all traffic to root path
#D Specifies the Python module path that Ray will import to find the application code
#E Number of replica vLLM instances
#F Ray downloads and extracts this repo as the working directory
#G vLLM is installed upon container startup
#H The application behavior is controlled using environment variables
#I The rest of the Ray cluster and worker group configuration.
```

It's completely understandable if you find the terminology a bit overwhelming at first, terms like "deployment," "service," or "instance" appear across Kubernetes, Ray, Ray Serve, and vLLM, often with different meanings depending on the context. To make this more intuitive, we encourage you to explore the hands-on example in the book's GitHub repo (<https://github.com/mllops-on>

[kubernetes/Book/blob/main/Chapter%208/vllm/vllm-ray/index.md](https://kubernetes.io/docs/tutorials/llm/vllm-ray/)), where we deploy a Mistral 7B model using Ray and vLLM. Walking through the actual deployment will help demystify how these layers work together in practice.

When deploying this solution in production environments, ensure that you allocate resources keeping Ray's architecture in mind. Ray's distributed nature means it works best with homogeneous GPU configurations across the cluster, and networking becomes a critical factor in performance. For optimal results, high-speed interconnects like RDMA or NVLink should be used between nodes handling distributed inference, as the tensor parallelism approach requires significant data transfer between GPUs during computation. Additionally, for fault-tolerant Ray clusters, consider using an external Redis cluster as discussed in Chapter 6. This helps ensure that the system can recover from node failures without losing the cluster's global state.

8.6 *Embracing the ecosystem*

There's a reason we didn't call this chapter "Running vLLM on Kubernetes" even though we admit keeping vLLM in focus throughout the chapter. It's important to recognize that the LLM serving is diverse and rapidly evolving. To remain competitive, you should also familiarize yourself with alternatives like Hugging Face's Text Generation Inference (TGI) and NVIDIA's AI Enterprise stack. The core concepts we've covered, paged attention, continuous batching, KV cache management, and distributed inference, are foundational across all modern LLM serving frameworks. Your experience with vLLM provides a strong conceptual foundation that will transfer readily to these alternative solutions. If tomorrow TGI offers better performance for your specific models or NVIDIA's tooling provides optimizations for your hardware configuration, you'll be well-equipped to make the switch without starting from scratch.

8.7 *Cleanup*

Ensure that you've deleted any resources created during this chapter so you're no longer paying for expensive GPU nodes.

To check if you have any vLLM deployments, run:

```
$ helm ls
```

If there are any results, uninstall by running `helm uninstall <release-name>`. For instance:

```
$ helm uninstall vllm
```

Check if there are any RayServices running in your cluster and delete them.

```
$ kubectl get rayservices  
$ kubectl delete rayservices <name>
```

Summary

- LLMs require specialized optimizations for memory management, KV cache handling, and efficient token generation that standard ML serving frameworks don't address.
- Purpose-built inference servers like vLLM, NVIDIA NIM, and TGI optimize LLM workloads with features tailored for transformer architecture efficiency.
- vLLM improves LLM inference efficiency through Paged Attention and continuous batching while providing OpenAI-compatible APIs.
- Official Helm charts and production-stack configurations enable standardized deployment of vLLM in Kubernetes environments.
- The two-tier architecture (routers and engines) separates concerns for optimal scaling and provides session-aware routing to maximize KV cache reuse.
- vLLM supports both online API-based serving and offline batch inference through simple CLI or Python interfaces.
- vLLM can serve multiple models through a single API endpoint, routing requests to the appropriate engine instance based on the model name.
- LoRA adapters enable efficient serving of specialized model variants without duplicating the underlying base model weights.
- Use Kubernetes rolling updates with appropriate readiness probes for zero-downtime model deployments.
- Load models that are cached using Persistent Volumes for faster container startup and scaling.
- InitContainers can download models from cached storage before the main engine container starts.
- An in-cluster storage can further reduce redundant model downloads.
- Techniques like pre-pulling or node image customization can reduce or eliminate container image pull latency.

- Custom metrics like latency, tokens per second, and requests per second provide better autoscaling signals than standard CPU/memory metrics for GPU workloads.
- Test throughput, latency, and resource utilization to identify optimal configurations for cost and performance.
- Ray integration enables scaling vLLM across multiple nodes with built-in load balancing and fault tolerance.

9

Observing ML Applications in Kubernetes

This chapter covers:

- Implementing observability in ML systems
- Collecting telemetry with OpenTelemetry
- Scraping metrics using Prometheus
- Custom scaling ML workloads
- Visualizing metrics with Grafana
- Profiling ML performance bottlenecks

Observability, an awfully pompous-sounding word borrowed from control theory, is simply your application's way of whispering (or screaming, depending on your logging level) exactly what's going on beneath the hood. It's the collective machinery that lets you answer questions you didn't even know you'd need to ask at three in the morning, when something's gone terribly wrong, and your system is suddenly behaving like a particularly mischievous raccoon rifling through your Kubernetes cluster.

I've been deliberately kicking this can down the road, not out of negligence, but out of respect. Observability is a complex topic that deserves its own limelight. In this chapter, we discuss two key aspects: how to implement observability in ML applications and how to effectively collect, visualize, and leverage this critical data at runtime. By the end of this chapter, you'll know what it takes to improve observability of a system and the tools of trade.

Finally, we'll analyze each component we've used in our system, from JupyterHub to vLLM, and review observability best practices.

Together, these elements will empower you to diagnose, debug, and even predict the health of your Kubernetes ML applications with confidence and clarity, ensuring fewer service interruptions and more restful nights.

9.1 Understanding Observability

Observability fundamentally revolves around three core pillars: logs, metrics, and traces. Logs provide detailed records of discrete events within your application, serving as your first line of defense when troubleshooting. Metrics are quantifiable data points that help measure performance and resource usage over time, offering insights into application health. Traces track the journey of a single request as it navigates through various system components, highlighting potential bottlenecks or latency issues, particularly in microservices architectures.

The particulars of what you log, what metrics you track, and whether you bother with tracing at all, those depend on your use case. Not every application needs all three pillars. Some workloads just need logs and a few key metrics to stay healthy. Others, especially in distributed or microservices environments, benefit immensely from tracing because it lets you see how a request meanders through the system and where the lag gremlins like to hide.

Observability becomes invaluable for complex systems, especially ML inference pipelines deployed as microservices. When a user request travels through multiple stages, such as tokenization, model inference, and post-processing, you must clearly understand the performance and interaction of each stage. Without tracing, pinpointing bottlenecks becomes a tedious game of log detective with incomplete clues.

9.1.1 New challenges with observing containers

The isolation model provided by containers, amplified by the adoption of microservices, has introduced new challenges for monitoring applications. Logging, monitoring, and tracing are by no means new concepts, but the shift to containers and microservices meant traditional methods no longer suffice. The ephemeral nature of containers, combined with the distributed deployment patterns inherent to Kubernetes, means applications could be scattered across multiple instances that frequently change locations.

This fluid, highly dynamic environment required new approaches to observability. Traditional monitoring solutions struggle with the transient lifecycle of containers and the complexity introduced by microservices architectures. As a result, the industry has seen the emergence of container-native tools designed explicitly to gather information effectively from ephemeral sources, consolidating vast amounts of data into coherent, actionable insights. These modern observability tools ensure you and I can still confidently make sense of this rich and constantly shifting operational landscape.

9.1.2 Observability Standards in Kubernetes

While Kubernetes doesn't ship with a full-featured observability suite, it does establish strong conventions, standards that most tools and platforms build upon.

Logging in containers is deceptively simple. The current de facto standard is: write to `stdout`. That's it. The container runtime, usually `containerd`, captures those streams and writes them to a file on the worker node. Kubernetes itself doesn't manage log aggregation, but it does expose the logs (via `kubectl logs`), and it expects you to bring a log collector to ship logs to a backend like Elasticsearch, Loki, or CloudWatch.

Metrics are typically exposed via an HTTP endpoint, most commonly `/metrics`, in a Prometheus-compatible text format. This convention makes it easy for a metrics collector tool like Prometheus to scrape metrics from your app (and your worker nodes and the cluster itself).

Tracing is less standardized out of the box, but Kubernetes makes it possible through annotations, sidecars, and open instrumentation standards like OpenTelemetry. Tools like Jaeger or Tempo can be integrated with your cluster to collect and visualize traces, but you'll need to make sure your application is instrumented appropriately.

9.1.3 Three Layers to Monitor

When building observability into a Kubernetes-based system, it helps to think in terms of layers. There are three primary layers to monitor:

- 23.The Kubernetes Cluster – This includes the control plane components such as the API server, etcd, controller manager, and scheduler. Monitoring this layer is about tracking the health and performance of the orchestration system itself. If your control plane becomes overloaded or misbehaves, your workloads might not even get scheduled.
- 24.The Worker Nodes – These are the actual machines (virtual or bare metal) where your containers run. You'll want visibility into CPU, memory, disk, and GPU utilization here. Tools like `node-exporter` and the `metrics-server` expose this data, and `kubelet` also provides runtime-level stats.
- 25.The Workloads – These are your applications: the ML models, inference services, training jobs, batch pipelines. This layer is where application-level metrics and logs live, and where request traces provide context. It's also the most diverse and dynamic part of the stack, which is why instrumentation needs to be deliberate and tailored to the specifics of each workload.

A good observability setup doesn't just focus on one of these layers, it connects all three. When something goes wrong, you need to be able to trace the issue from symptoms in your application all the way down to resource constraints or control plane hiccups.

OBSERVING KUBERNETES CONTROL PLANE - LOGS

Kubernetes, like all good cloud native applications, produces comprehensive logs and metrics to give you a detailed insight into its inner workings at all times. Remember those Kubernetes controllers that run in the control plane? The Kubernetes API server, Scheduler, and Controller Manager. They produce logs that are most vital when troubleshooting issues with the control plane. We're talking about things like:

- API Server logs – These are the cluster's diary entries. Every API call, every client interaction, every "who touched what and when" gets recorded here. If someone's complaining they can't create a pod, or kubectl is "too slow", this is your first stop.
- Scheduler logs – When your Pods are sitting around twiddling their thumbs in "Pending" state instead of running, the scheduler logs are the ones telling you why. Not enough resources? Node taints? Fat-fingered affinities? It'll be in here.
- Controller Manager logs – This guy is the puppet master. These logs tell you when Deployments, ReplicaSets, Jobs, or any other objects are being created, scaled, or deleted. If Kubernetes seems like it's playing a weird game of "now you see it, now you don't" with your workloads, these logs are where you start sniffing around.

Each control plane component leaves a paper trail. Your job is knowing where to find it when the cluster starts acting up.

In most cloud environments, enabling control plane logging is as simple as flipping a switch. Check your cloud provider's documentation to learn about the control plane controller logs available in your environment. We highly recommend you turn logging on for all your clusters, but at the very least, do it for production. Some teams try to save on costs by skipping logging for dev or test clusters, which is an acceptable tradeoff if you're operating at scale and watching your budget. Just don't make the mistake of flying blind in production.

OBSERVING KUBERNETES CONTROL PLANE - METRICS

Besides logs, the control plane also emits a steady stream of metrics in Prometheus format (<https://github.com/prometheus/docs/blob/main/content/docs/instrumenting/e>

xposition_formats.md). These metrics track everything from API request latencies to scheduler queue lengths to etcd database health. They're machine-readable and structured specifically so tools like Prometheus can scrape, store, and help you query them later.

If you want to know how many API calls are succeeding versus failing, how overloaded your scheduler is, or how much etcd disk I/O you're chewing through, control plane metrics have your answers. These numbers are less colorful than logs but no less important, they tell you if trouble is brewing long before users start noticing anything wrong.

In most cloud environments, enabling control plane metrics is straightforward. For instance, in Google Kubernetes Engine (GKE), you can enable these metrics via the Google Cloud Console or the gcloud CLI. Similarly, Amazon EKS allows you to fetch control plane metrics in Prometheus format.

The advantage of emitting metrics in Prometheus format is the flexibility it offers. You can integrate these metrics with your preferred monitoring tools, whether that's your cloud provider's native monitoring solution or open source tools like Prometheus and Grafana. This standardization ensures that you're not locked into a specific vendor and can choose the tools that best fit your operational needs.

For more detailed information on the metrics exposed by Kubernetes components, refer to the Kubernetes Metrics Reference: (<https://kubernetes.io/docs/reference/instrumentation/metrics>).

OBSERVING KUBERNETES DATA PLANE

While the control plane orchestrates the cluster, the real action happens in the data plane. Worker nodes are the workhorses executing your applications, which means monitoring them is vital for maintaining cluster health.

Worker nodes run essential components like kubelet and container runtime, which generate logs and metrics. They provide insights into node performance, resource utilization, and potential issues affecting your workloads.

In cloud environments, worker nodes are often ephemeral, they can be terminated, replaced, or scaled based on demand. This transient nature poses challenges:

- Log Volatility: Logs stored locally on a node can be lost when the node is decommissioned.
- Troubleshooting Difficulties: Without retained logs, diagnosing past issues becomes nearly impossible.

To mitigate this, it's crucial to implement centralized logging solutions that aggregate and store logs independently of the nodes.

On the flip side, while retaining logs is essential for observability and compliance, it comes with storage costs. High-resolution logs and extended retention periods can lead to significant expenses.

To strike the right balance:

- Define Retention Policies - Determine how long logs need to be retained based on compliance and operational requirements.
- Implement Log Rotation - Rotate logs to prevent disk space exhaustion.
- Filter Logs - Collect only necessary logs to reduce volume.
- Archive Strategically - Move older logs to cost-effective storage solutions.

On a side note, congratulations! You now have one more eternal question to answer: to retain or not to retain? It's a bit like figuring out what video resolution to save your family memories in, you want enough detail to cherish the moment, but not so much that you drown in storage costs. Welcome to the glamorous life of a Kubernetes architect.

OBSERVING KUBERNETES APPLICATIONS

Now let's turn our attention to the workloads themselves, the real stars of the show. Observing ML applications running in Kubernetes isn't a guessing game. It follows well-established cloud-native patterns, and the first principle is this: if you want to observe something, you have to instrument it.

Instrumentation refers to adding logic, manually or automatically, that allows your application to emit signals about what it's doing. These signals typically fall into three categories: logs, metrics, and traces. Tools like OpenTelemetry (which we'll explore shortly) make this easier by offering both SDKs and zero-code instrumentation that hook into the language runtime, your web framework, and your model server. The result is a steady stream of telemetry data that can be sent to your backend of choice, whether that's Prometheus, Grafana Tempo, Jaeger, or a managed observability platform.

Instrumentation is the code you add to your application to capture information about what it's doing at runtime. Tools like OpenTelemetry (we'll meet it very soon) make it easier by providing libraries that automatically gather data from the language runtime, the framework you're using, and your application logic. This data, metrics, logs, and traces, is then sent to your observability backend of choice, whether that's a Prometheus instance, a Jaeger server, or a cloud provider's monitoring system.

Let's start with logs. In Kubernetes, the expectation is simple: write to `stdout`. That's it. Don't write logs to random files. Logging agents like Fluent Bit

and the OpenTelemetry Collector are built to capture these standard streams. And when you log, structure your logs, JSON is a solid choice. It makes downstream parsing, filtering, and querying significantly easier.

Next are metrics. Cloud-native applications should expose metrics in a format that monitoring tools can easily scrape, typically, Prometheus exposition format over an HTTP endpoint. OpenTelemetry (or Prometheus) can collect a lot of runtime-level metrics out of the box, but you'll likely want to define custom ones as well. The good news is that adding a metric is often just a few lines of code in the right place.

Then there's tracing. Tracing provides a request-centric view of how operations unfold across services. In ML systems, this is particularly important when your model is part of a broader service graph, think feature stores, tokenizers, preprocessors, postprocessors, and downstream APIs. When something is slow, a trace tells you not just that it's slow, but where it's slow.

If you're deploying models in Kubernetes, a good design pattern is to treat each model as a microservice, an idea we already discussed with Ray Serve and vLLM earlier in the book. These frameworks often support tracing natively or using a middleware.

And no, you don't have to hand-craft every tracing header or write your own instrumentation from scratch. Modern frameworks do most of the heavy lifting. The emerging standard that ties all of this together (logs, metrics, and traces) is OpenTelemetry.

9.2 *Observing with OpenTelemetry*

OpenTelemetry, often lovingly referred to as "OTEL" by folks who have spent too much time in YAML files, is a CNCF project that provides a standard way to generate, collect, and export telemetry data. It has two main parts:

- Producers (SDKs) – Libraries embedded into your applications that create and export telemetry data: logs, metrics, and traces.
- Collectors – Standalone agents or services that receive telemetry data and forward it to your backend of choice (like Jaeger, Grafana Tempo, Prometheus, or your cloud provider's observability service).

Before OpenTelemetry, teams had to bolt together separate tools to handle logs, metrics, and traces—each with its own collectors, exporters, and configuration headaches. Fluentd or Fluent Bit for logs, Prometheus for metrics, and something else for traces.

OpenTelemetry changes that. It gives you one framework to do all three (because "I want three tools when one will do the job," said no one ever.) One

collector. One set of SDKs. One unified way to handle observability data from generation to export.

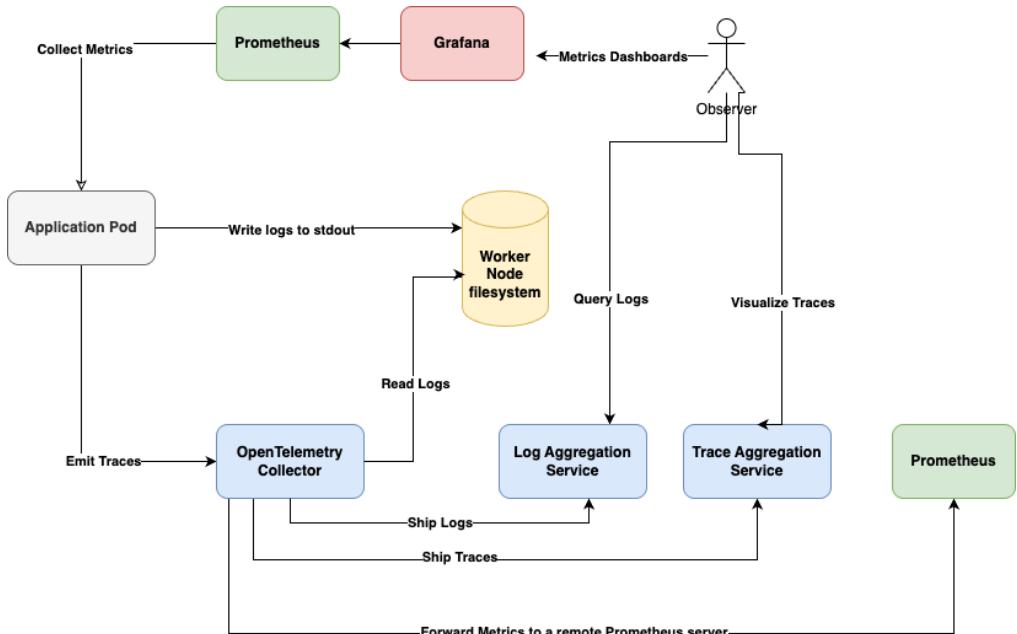


Figure 9.1 OpenTelemetry collects logs, metrics, and traces produced by workloads running in a Kubernetes cluster. If there's no Prometheus server in the cluster, the OpenTelemetry Collector

When it comes to machine learning applications deployed on Kubernetes, this separation is pure gold. You can focus on emitting data from your app (producers) and let a robust, configurable collector deal with batching, retrying, and sending it somewhere useful.

Most ML applications today are heavy on Python, with maybe a sprinkling of Node.js for frontends. In these environments, standard cloud-native logging best practices (log to stdout, structure your logs) are usually more than enough.

As for metrics and traces? Frameworks like Ray, FastAPI, and vLLM expose Prometheus-style metrics and produce traces natively or via extensions. Ergo, you won't need to hand-roll instrumentation for every little thing.

The biggest advantage of OpenTelemetry is its flexibility. You can switch where your telemetry data goes, say, from self-managed to a cloud-native

observability service, without changing a single line of application code. All you must do is reconfigure the collector.

Observing ML Applications

Observing ML applications requires more than just watching for 500 errors and CPU spikes. ML workloads bring new dimensions to observability, dimensions like data drift, concept drift, model accuracy degradation, and training instability. These are not your average web app problems.

To observe ML systems effectively, you need to track things like:

- Model performance metrics – Accuracy, precision, recall, F1-score, or any domain-specific KPIs you care about. This can be surfaced from a post-inference logging component or monitored continuously with shadow deployments.
- Data distributions – Feature-level stats that let you detect when the input distribution has shifted from training.

It's worth noting that model-specific metrics like accuracy or F1-score are generally static—they're logged during training and stored in model registries or experiment trackers like MLflow. These aren't what we focus on in this chapter, since they fall more under the umbrella of model management than runtime observability.

Things like model drift and feature attribution, on the other hand, are relevant here. They are a runtime concern. Monitoring models includes tracking both what the model is predicting and why it's making those predictions. The simplest way to monitor model's health and performance is to periodically re-evaluate the model against a known validation dataset or to compare predictions with real outcomes when ground truth becomes available later.

9.3 *Collecting Logs with OpenTelemetry*

In Kubernetes, the design choice to write logs to `stdout` is intentional, it allows the platform, and the tools that support it, to collect logs in a consistent and scalable way. Once your application emits logs, the next step is to collect, process, and store them for later analysis. This is done by the Collector.

The *OpenTelemetry Collector* is a high-throughput, vendor-neutral service designed to receive telemetry data (logs, metrics, and traces), apply optional processing, and forward it to a backend system. For logs specifically, it integrates with the container runtime (such as `containerd`) to capture logs directly from running containers. This eliminates the need to modify your applications or write log files manually.

One of the key strengths of the Collector is its flexibility. It can filter, transform, and enrich logs before exporting them. It can standardize formats to match the requirements of your destination system. And it can add consistent metadata (like Kubernetes labels or trace context) so that logs, metrics, and traces can be correlated more effectively across the entire system. This correlation is also valuable in distributed ML systems, where components may be developed and deployed independently.

In most cloud-native environments, logs are sent to a centralized platform that supports search, visualization, anomaly detection, alerting, and long-term storage. Common destinations include Elasticsearch, Grafana Loki, or cloud-specific services like AWS CloudWatch. The Collector acts as the central point for routing logs efficiently and reliably.

In Kubernetes, the OpenTelemetry Collector is most commonly deployed as a DaemonSet, with one instance running on each node to collect local logs. In some cases, it can also be deployed as a sidecar or a standalone service, depending on your architecture.

The configuration of the Collector is composed of four main parts:

- Receivers – Define how data is ingested (e.g., from logs, Prometheus, OTLP).
- Processors – Modify or filter data as it moves through the pipeline.
- Exporters – Define where data is sent (e.g., CloudWatch, Tempo, Elasticsearch).
- Service / Pipelines – Combine receivers, processors, and exporters into end-to-end workflows.

In practice, most teams install the Collector once, configure it according to their observability needs, and let it run with minimal ongoing maintenance. Of course, like any part of your infrastructure, it should be monitored, but once it's in place, it provides a robust, scalable foundation for log collection that works seamlessly across your Kubernetes environment.

Next, let's explore the second pillar of observability: metrics. While log collection is generally a passive operation, capturing output as it happens, Kubernetes is designed to take a much more active role in how metrics are produced, scraped, and used. Metrics are not just for visibility; they are often

used to drive decisions, such as autoscaling and alerting, making them a critical part of running reliable, responsive systems.

9.4 *Introducing Prometheus*

Prometheus, the open-source monitoring system now housed under the CNCF (Cloud Native Computing Foundation), is the most widely adopted tool for collecting metrics in Kubernetes environments. Its name isn't accidental. In Greek mythology, Prometheus was the Titan who stole fire from the gods and gave it to humanity, an act of rebellion that brought knowledge, light, and civilization. In the same spirit, the Prometheus monitoring system gives you the ability to monitor your systems, not by guesswork, but by hard, quantifiable data.

At its core, Prometheus is a time series database. It ingests metrics data over a period. Things like CPU usage, memory consumption, number of HTTP requests, model inference latency, token generation rates.

Prometheus collects this data by scraping HTTP endpoints (it pulls data by default) that expose metrics in a specific format (the "Prometheus exposition format"). Each metric it ingests is stored with a set of labels, allowing powerful querying and filtering later.

It's important to understand what Prometheus does and doesn't do. Prometheus focuses solely on metrics. It doesn't store logs. It doesn't collect traces. It's not trying to be everything. It's built to be excellent at one thing: making time series metrics easy to collect, query, and alert on.

If you've been poking around as we built our MLOps platform on Kubernetes, you may have noticed a pattern: every open-source tool we've touched (Kubeflow, Ray, vLLM, Airflow, even the Kubernetes control plane itself) exposes metrics in Prometheus format. This isn't an accident. It's the lingua franca of observability in modern cloud-native systems. If you're building MLOps on Kubernetes, getting familiar with Prometheus a necessity.

Prometheus complements OpenTelemetry. OpenTelemetry focuses on producing signals (logs, metrics, and traces) from applications. Prometheus focuses on pulling in those metrics (particularly anything exposed in Prometheus format) and storing them for analysis and alerting. In multi-cluster environments, you may have a centralized Prometheus server that receives metrics from each cluster. In this case, you won't install Prometheus in each cluster but configure OpenTelemetry Collector in each cluster to forward metrics to the central Prometheus server.

Once your metrics land in Prometheus, they become accessible to all sorts of tools like Grafana dashboards, custom alerting rules, and anomaly detection

pipelines. It's the foundation for making your ML workloads observable, measurable, and ultimately, reliable.

9.4.1 Prometheus Exposition Formats

Before Prometheus can scrape or receive metrics, they need to be structured in a way Prometheus understands. Most applications and services expose metrics in Prometheus text exposition format.

In Metrics are exposed as plain text over HTTP, usually at a /metrics endpoint. Each metric is a line of text showing the metric name, optional labels, and its value. For instance, the Kubernetes cluster exposes its metrics in this format. You can view these metrics by running `kubectl get --raw /metrics` in your cluster.

When you run this command, you'll see a stream of text that looks something like this:

```
# HELP apiserver_watch_events_total [ALPHA] Number of events
sent in watch clients
# TYPE apiserver_watch_events_total counter
apiserver_watch_events_total{group="",kind="ConfigMap",version="v1"} 174498
```

Each metric starts with optional HELP and TYPE comments that describe what the metric measures. Then you'll see the actual metric values: a metric name, optional labels in {} brackets, and a numerical value.

Prometheus supports a few core metric types, each suited for a different kind of measurement:

- Counter – A cumulative metric that only ever increases (or resets). Think of things like the total number of HTTP requests received or the total number of model inferences completed.
- Gauge – A metric that can go up or down. Gauges are perfect for tracking values like current memory usage, GPU temperature, or the number of active requests.
- Histogram – Measures the distribution of events over configurable buckets. Histograms are great when you want to understand things like request durations or payload sizes (how many were fast, how many were slow, and everything in between).
- Summary – Similar to a histogram but focuses on providing calculated quantiles (like 95th percentile latency) directly from the client side, without needing post-processing on the Prometheus server.

When you look at the `# TYPE` line in the exposition format (like `# TYPE apiserver_watch_events_total counter`), it tells you which kind of metric you're dealing with and how you should interpret its behavior over time.

Now, most of the time, you're not going to define custom metrics—because the framework you're using is already generous with what it exposes. But every so often, you'll want to track something specific. Maybe it's a business metric. Maybe it's a metric that helps you scale smarter.

Remember when we talked about vLLM? We discussed scaling the engine pods horizontally based on the number of inflight requests, a metric that came out of the box. But in your setup, you might have an even better signal. Or maybe you just need to record something custom for debugging, alerting, or reporting.

When that moment comes, take a second to think about what kind of metric it really is. Is it a counter? A gauge? A histogram? Picking the right type isn't just bookkeeping, it shapes how you'll interpret and use that metric down the line.

9.4.2 Choosing a monitoring system for your workloads

Prometheus has become the de facto standard for metrics collection in cloud-native environments. And that's a good thing. It means you're not locked into any one toolchain. You can use whatever observability backend you prefer (for example, Amazon CloudWatch, Datadog, Grafana Cloud) and nothing in your cluster needs to change. All you do is point your OpenTelemetry agent or collector to send metrics somewhere else.

In fact, most major cloud observability platforms now natively support ingesting metrics in Prometheus format. AWS, Google Cloud, Azure, Datadog, Grafana Cloud, all of them can scrape or receive Prometheus-style metrics without needing translation layers. Some of these platforms even offer fully managed Prometheus backends, so you don't have to run Prometheus yourself.

When setting up Prometheus, most teams choose one of two approaches:

- Deploy Prometheus inside the cluster – Set up a Prometheus server directly in your Kubernetes cluster to scrape and store metrics locally. This is simple, fast, and gives you immediate visibility with minimal moving parts.
- Use OpenTelemetry with Remote Write – Configure your OpenTelemetry collector to forward metrics to an external Prometheus-compatible backend. This setup shines when you're aiming for centralized observability across multiple clusters or cloud accounts.

Most people start simple: they run an in-cluster Prometheus because it gets the job done. Later, when you have multiple Kubernetes clusters (and suddenly

more dashboards than you can count), you graduate to a centralized observability stack.

You don't have to get it perfect on Day One, just get visibility first and scale your observability stack as your platform grows.

9.4.3 *Installing Prometheus in Your Cluster*

You don't need to install Prometheus to follow along with the rest of this chapter, but if you want to get your hands dirty, it's easy to spin up a full observability stack in your own cluster.

We'll use the kube-prometheus-stack (<https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack>), a well-supported and opinionated Helm chart that bundles everything you need:

- Prometheus
- Prometheus Operator
- kube-state-metrics
- Node Exporter
- Alertmanager
- Grafana
- And a set of pre-wired dashboards and alerts

To install the kube-prometheus-stack, run:

```
$ helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts
$ helm repo update prometheus-community

cd "Chapter 9/kube-prometheus-stack"

helm install kube-prometheus-stack prometheus-community/kube-
prometheus-stack \
  --version "48.1.1" \
  --namespace monitoring \
  --create-namespace \
  -f kube-prometheus-values.yaml
```

Give it a minute or two, then check your pods:

```
$ kubectl get pods -n monitoring
```

You should see output similar to this:

NAME	READY
STATUS	
kube-prometheus-stack-grafana-68fd775d7f-ss2zn	3/3

```

Running   0
kube-prom-stack-kube-prome-operator-7886bd7fb5-fdjr9  1/1
Running   0
kube-prom-stack-kube-state-metrics-59856b99cb-j697d  1/1
Running   0
kube-prom-stack-prometheus-node-exporter-55462        1/1
Running   0
prometheus-kube-prom-stack-kube-prome-prometheus-0     2/2
Running   0

```

Once all Pods are running, you can access Prometheus UI to see which metrics are currently being scraped:

```
$ kubectl port-forward svc/kube-prom-stack-kube-prome-prometheus
9090:9090 -n monitoring
```

Then open a browser to `http://localhost:9090`. Head to Status → Targets, and you'll see exactly which endpoints Prometheus is scraping and how they're doing.

The screenshot shows the Prometheus UI Targets page. At the top, there are tabs for All, Unhealthy, and Expand All, with All selected. Below that is a search bar and filter options for Unknown, Unhealthy, and Healthy status. The main area displays three service monitors:

- serviceMonitor/kube-system/gpu-operator/0 (1/1 up)**: This monitor has one endpoint at `http://10.0.31.186:8080/metrics`, which is UP. The labels for this endpoint are: container="gpu-operator", endpoints="gpu-metrics", instance="10.0.31.186:8080", job="gpu-operator", namespace="kube-system", pod="gpu-operator-5694b4c6568-mtkam", service="gpu-operator".
- serviceMonitor/kube-system/nvidia-dcgm-exporter/0 (1/1 up)**: This monitor has one endpoint at `http://10.0.29.242:9400/metrics`, which is UP. The labels for this endpoint are: container="nvidia-dcgm-exporter", endpoints="gpu-metrics", instance="10.0.29.242:9400", job="nvidia-dcgm-exporter", namespace="kube-system", pod="nvidia-dcgm-exporter-v895m", service="nvidia-dcgm-exporter".
- serviceMonitor/monitoring/kube-prom-stack-grafana/0 (1/1 up)**: This monitor has one endpoint at `http://10.0.29.242:9090/metrics`, which is UP. The labels for this endpoint are: container="grafana", endpoints="grafana-metrics", instance="10.0.29.242:9090", job="grafana", namespace="monitoring", pod="grafana-7445588666-6qk6t", service="grafana".

Figure 9.1 A screenshot of Prometheus UI showing targets from which this Prometheus server is pulling metrics. Each target has a one or more endpoints from which Prometheus scrapes metrics.

Now that we have the kube-prometheus stack up and running, let's take a closer look at everything it installed , and more importantly, what role each component plays in making Kubernetes observability work.

9.4.4 *Breaking Down the kube-prometheus-stack Components*

Let's go over the components of kube-prometheus-stack.

GRAFANA (KUBE-PROM-STACK-GRAFANA)

Grafana is your visualization layer. It's where you build dashboards, slice and dice metrics, and make pretty graphs that turn into 2 a.m. on-call alerts.

It connects directly to Prometheus as a data source and pulls whatever metrics you want to see. This pod typically runs three containers: Grafana itself, a config reloader that reloads Grafana configuration on the fly, and a sidecar that imports some prebuilt dashboards and datasources automatically.

Nothing complicated here. It's the UI that makes raw numbers actually useful.

PROMETHEUS OPERATOR (KUBE-PROMETHEUS-OPERATOR)

The Prometheus Operator is the brains behind the whole setup.

It watches the Kubernetes API for custom resources like ServiceMonitor (more on this soon), PodMonitor, Prometheus, and Alertmanager objects.

When you create or update any of these, the operator automatically reconfigures the actual Prometheus servers running in your cluster.

KUBE-STATE-METRICS (KUBE-STATE-METRICS)

`kube-state-metrics` is a metrics generator, not a scraper.

It watches the Kubernetes API and exports cluster state as metrics. It tells you things like how many pods are running, how many are pending, how many replicas your deployments have, and how much of your resource quotas you've burned through.

It doesn't monitor CPU or memory usage, that's Node Exporter's job. `kube-state-metrics` is purely about the health and state of Kubernetes objects.

NODE EXPORTER (PROMETHEUS-NODE-EXPORTER)

Node Exporter is deployed as a DaemonSet. That means there's one instance running on every node in your cluster, quietly scraping the node itself.

It gathers system-level metrics like CPU usage, memory usage, disk I/O, filesystem stats, and network throughput. It does this by reading from the Linux `/proc` and `/sys` filesystems. If you want to know whether your EC2 instance or

your on-prem node is having a bad day, Node Exporter is where that story gets written.

PROMETHEUS SERVER (KUBE-POME-PROMETHEUS-0)

Finally, the main event: the Prometheus server itself. It is deployed as a StatefulSet, not a regular Deployment.

That's because Prometheus needs stable network identities and persistent storage, it's not just spinning up stateless web servers.

Typically, this StatefulSet is backed by a block storage device (a PersistentVolumeClaim), where Prometheus writes its time series database (TSDB) to disk. This TSDB is where Prometheus stores all your scraped metrics: compressed, chunked, and optimized for fast reads.

If your Prometheus pod crashes, it restarts and reattaches to the same volume, no scraped metrics lost.

ALERTMANAGER

Alertmanager handles alert routing, deduplication, grouping, and notification dispatch for Prometheus alerts. It's designed to take firing alerts from Prometheus and transform them into notifications via various channels like email, Slack, PagerDuty, or custom webhooks.

However, it's worth noting that in our configuration, Alertmanager has been disabled in the values file we provided to the Helm chart , a common practice in development or learning environments where you want to focus on metric collection and visualization without setting up alerting pipelines. In production environments, you would typically enable Alertmanager to ensure that critical issues trigger appropriate notifications to your operations team.

When enabled, Alertmanager provides sophisticated features like silencing alerts during maintenance windows, throttling notifications to prevent alert storms, and routing different types of alerts to different teams or channels based on severity, service, or other labels.

9.5 *Target discovery in Prometheus*

We learned that Prometheus collects metrics by scraping target endpoints. So how does Prometheus even know which targets to scrape? In Kubernetes, Prometheus discovers scrape targets dynamically using Kubernetes service discovery. By default, Prometheus monitors the standard Kubernetes resources (like Pods, Services, Deployments). If you want it to monitor a custom workload

metric, then it requires additional configuration. Let's see this in action using vLLM as an example.

9.5.1 Understanding Service and Pod Monitors

When we talked about scaling vLLM deployments using custom metrics, you might remember a little thing called a Service Monitor. Let's zoom in on this as it is crucial for understanding how custom metric collection work in Prometheus.

A *Service Monitor* is a Kubernetes Custom Resource Definition (CRD) that tells Prometheus how to discover and scrape metrics from services in your cluster. Instead of manually configuring Prometheus with static targets, you define a ServiceMonitor CRD and Prometheus dynamically discovers and scrapes the right endpoints.

Similarly, a *PodMonitor* skips the Service entirely and scrapes metrics directly from Pods based on their labels. The key difference is simple: ServiceMonitors use Services as the entry point, while PodMonitors go straight to Pods.

NOTE You might notice that the full resource name for Service Monitor is servicemonitors.monitoring.coreos.com. That's because the Prometheus Operator was originally created by CoreOS (before they were acquired by Red Hat). Even though the project has moved into the broader Kubernetes and CNCF ecosystem, the CRD name has stayed the same for backward compatibility.

You can see all the ServiceMonitors in your cluster right now by running:

```
$ kubectl get servicemonitors -A
```

You'll likely find entries for things like the GPU Operator (gpu-operator-metrics) and the DCGM Exporter (dcgm-exporter-metrics). These Service Monitors are why GPU metrics show up in Prometheus without you having to manually point Prometheus at random pod IPs.

Each Service Monitor typically specifies:

- What namespace to look in
- Which services (by label selector) to scrape
- Which port and path to use for scraping (/metrics, usually)
- Optional TLS or authentication settings

Prometheus watches for Service Monitor resources and automatically updates its configuration behind the scenes. This decoupling means that as Pods come and go, or workloads scale, Prometheus keeps scraping the right things.

When we installed the kube-prometheus-stack earlier, it didn't just install Prometheus and Grafana. It also created a bunch of supporting Kubernetes Services that expose metrics endpoints for Kubernetes components like CoreDNS, kube-controller-manager, kube-proxy, kube-scheduler, etcd, and kubelet. Each of these Services is paired with a corresponding ServiceMonitor that tells Prometheus: "Go scrape this Service on this port at this path."

That's why, when you open the Prometheus UI and look at Status → Targets, you already see metrics flowing in, even though you never manually configured a single scrape target.

You can see these system services by running:

```
$ kubectl -n kube-system get services | grep kube-prom
```

Listing 9.1 contains an example of a vLLM Service Monitor (trimmed down for clarity). ServiceMonitor CRD defines which services to scrape and how (endpoints, path, port, interval). The Prometheus Operator watches ServiceMonitors and updates Prometheus configuration.

Listing 9.1 A Service Monitor to scrape vLLM metrics

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: vllm-monitor #A
  namespace: default
spec:
  selector:
    matchLabels:
      app: vllm-engine #B
  namespaceSelector:
    matchNames:
    - default #C
  endpoints:
  - port: service-port #D
#A Sets the name and namespace where the ServiceMonitor lives
#B Selects the Kubernetes Service to scrape, based on its labels.
#C Tells Prometheus which namespace(s) to look in for matching Services.
#D Specifies which named port on the Service to scrape metrics from.
```

After scraping, Prometheus stores metrics in a time series database and serves them up via the Prometheus query language (PromQL). These metrics can then be used by ops teams to monitor application health, often through custom

dashboards designed around the specific workload, and by developers to continuously improve performance.

As an MLOps engineer, you'll use these metrics any time you want Prometheus to start collecting metrics from your own workloads. Whether it's a model inference service, a training job with custom exporters, or a batch job emitting stats.

9.6 Scaling based on custom metrics

Prometheus not only allows you to observe your system through application-specific metrics (like the number of inference requests handled, model queue depth, or token generation rates) but also provides visibility into infrastructure metrics such as CPU, memory, and disk utilization. But observability isn't the end goal. Once we can measure something, the natural next step is: can we respond to it automatically?

In cloud-native systems, one of the most common and effective responses to changing metrics is autoscaling. When traffic increases, we want our system to scale out and meet demand. When load decreases, we want to scale in and reduce cost. Prometheus gives us the foundation to enable this kind of adaptive behavior.

Scraping metrics is just the first step. The next is being able to act on those metrics. For example, what if your vLLM inference pods start handling a surge in requests per second? You may want to automatically spin up more pods to maintain low latency.

To make that possible, you first expose your workload's metrics in Prometheus format, then wire them into Prometheus using a Service Monitor. Once the metrics are flowing into Prometheus, you're halfway there.

This is where the *Prometheus Adapter* enters the picture. It acts as a bridge between Prometheus and Kubernetes' Horizontal Pod Autoscaler (HPA). Specifically, it exposes selected Prometheus metrics as custom metrics in the Kubernetes API. This lets you scale workloads based on whatever matters most for your application, not just CPU or memory.

To install Prometheus Adapter in your cluster, run:

```
$ helm install prometheus-adapter prometheus-  
community/prometheus-adapter \  
--namespace monitoring \  
-f prom-adapter.yaml
```

After a few minutes, you'll be able to see custom metrics being exposed in your cluster currently by running:

```
$ kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1/
```

This is your first look into the Custom Metrics API group (`custom.metrics.k8s.io`) (<https://github.com/kubernetes-sigs/custom-metrics-apiserver>). This API group isn't part of core Kubernetes, it's an extension, created by the Prometheus Adapter. The Prometheus Adapter produces this API by translating scraped Prometheus metrics into a format the Kubernetes API server understands. Kubernetes controllers like the HPA consume it. When an HPA is configured to scale on a custom metric (e.g., vLLM number of requests waiting), it queries this API to determine the current metric value.

You can think of the adapter as an API translator. It takes the Prometheus query language (PromQL) and re-expresses the selected metrics through an API that Kubernetes natively understands. This way, the autoscaling machinery doesn't need to know anything about Prometheus, it just knows how to talk to an API endpoint.

By default, the Prometheus Adapter won't expose every metric Prometheus has. You must configure rules in the adapter's values file (e.g., `prom-adapter.yaml`) to specify which metrics should be surfaced and under what names. This gives you fine-grained control, only metrics that are explicitly selected will be exposed.

Here's the high level flow:

26. You export a Prometheus metric from your app (say, `vllm_num_requests_waiting`).
27. Prometheus scrapes it using a ServiceMonitor.
28. The Prometheus Adapter maps that metric into the Kubernetes Custom Metrics API.
29. The HPA queries that API to make scaling decisions.
30. If required, the HPA adjusts the underlying Kubernetes Deployment's replica count.

The result is that you get autoscaling behavior based on the exact metric that reflects application demand. Let's see this action.

9.7 Monitoring and scaling an application

We've spoken in abstract long enough and now we must complete our learning by getting our hands dirty. We'll now create a vLLM deployment in our cluster, expose its custom metrics, and configure autoscaling.

To begin, let's create a vLLM deployment like we did in the previous chapter:

```
$ helm install vllm vllm/vllm-stack -f vllm-example.yaml
```

NOTE Deploying this chart will create worker nodes with Nvidia GPUs that can be quite pricey. Please be aware of the cost implications of running this workload.

When we deploy vLLM, it creates two components: a router and an engine. By default, both start with a single replica. Our goal is to dynamically adjust the number of engine replicas in response to traffic patterns. More specifically, we want Kubernetes to scale the engine Pods based on the number of pending inference requests.

vLLM automatically exposes metric `vllm_num_requests_waiting` that indicates how many requests are currently queued. When this number grows, it suggests that users are experiencing increased latency. By scaling out, adding more engine replicas, we can reduce this queue, thereby improving responsiveness and overall user experience.

Ideally, we'd scale up indefinitely, assigning a dedicated engine Pod to each request. But in reality, compute resources are limited and costly. The challenge, then, is to find the right balance: one where users receive fast responses, but the system remains efficient and cost-effective to operate. This is where custom autoscaling becomes a powerful tool.

Now that we have vLLM running in our cluster. We need two more things to monitor and scale it: a Service Monitor to scrape this vLLM metric and a Prometheus Adapter to put that metric into the Kubernetes Custom Metrics API.

9.7.1 *Creating a Service Monitor*

When we deployed the vLLM Helm chart, we also added a custom label (`app=vllm-engine`). Now we can use this label to create a Service Monitor that scrapes the vLLM metric we're interested in. Listing 9.2 contains the definition of the Service Monitor that looks for a Kubernetes Service with label (`app=vllm-engine`) on port 8080 (it is defined as the "service-port" in the vLLM Deployment).

Listing 9.2 Service Monitor for scraping vLLM metric

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: vllm-monitor #A
  namespace: default
spec:
```

```

selector:
  matchLabels:
    app: vllm-engine #B
namespaceSelector:
  matchNames:
    - default #C
endpoints:
  - port: service-port #D
#A The name of this Service Monitor
#B The label the Service Monitor uses for selecting one or more Services
#C The namespace in which the Service to be monitored exists
#D The port on which the metrics are exposed. In this case the metrics endpoint is the same as the service endpoint.

```

Create the Service Monitor by running:

```
$ cd cd kube-prometheus-stack
$ kubectl apply -f vllm-servicemonitor.yaml
```

After creating this Service Monitor, Prometheus will start scraping the metric. When we deployed the Prometheus Adapter, we created a rule to expose the `vllm:num_requests_waiting` metric to the Kubernetes Custom Metrics API. Now, we can see that Prometheus is ingesting this metric and exposing it using Kubernetes API, we can use `kubectl` to get the value for this metric. To get metric's current value, run:

```
$ kubectl get --raw
/apis/custom.metrics.k8s.io/v1beta1/namespaces/default/metrics/v
llm_num_requests_waiting | jq
{
  "kind": "MetricValueList",
  "apiVersion": "custom.metrics.k8s.io/v1beta1",
  "metadata": {},
  "items": [
    {
      "describedObject": {
        "kind": "Namespace",
        "name": "default",
        "apiVersion": "/v1"
      },
      "metricName": "vllm_num_requests_waiting",
      "timestamp": "2025-05-07T17:41:07Z",
      "value": "0",
      "selector": null
    }
  ]
}
```

We can see that the current value is "0", which means vLLM has no requests waiting currently. To verify that Prometheus is now scraping vLLM successfully,

head to the Prometheus UI (Status → Targets) and look for an entry related to vllm-monitor. If it's marked as UP, you're good to go.

9.7.2 Exposing metrics using Prometheus Adapter

A key detail we skipped discussing previously was how we configured the Prometheus Adapter to expose the vLLM metric. When we installed the Prometheus Adapter, we created a rule using the Helm chart to expose the vllm:num_requests_waiting metric to the Kubernetes Custom Metrics API.

The adapter's configuration consists of rules that define which Prometheus metrics should be exposed as Kubernetes custom metrics. Each rule includes:

- Discovery - A seriesQuery that finds metrics matching a specific pattern
- Association - Mapping between Prometheus labels and Kubernetes objects
- Naming - How the metric will be identified in the Kubernetes API
- Querying - How to fetch the metric value when Kubernetes requests it

The rule shown in listing 9.3 tells the adapter to:

31. Find metrics named exactly vllm:num_requests_waiting
32. Associate them with Kubernetes namespaces
33. Expose them as vllm_num_requests_waiting in the Kubernetes API
34. Use a PromQL query to calculate the value per namespace

Listing 9.3 seriesQuery for exposing a vLLM metric

```
- seriesQuery: '{__name__ =~ "vllm:num_requests_waiting$"}'  
  resources:  
    overrides:  
      namespace:  
        resource: "namespace"  
    name:  
      matches: ""  
      as: "vllm_num_requests_waiting"  
    metricsQuery: sum by(namespace) (vllm:num_requests_waiting)
```

For more detailed explanation of the adapter's configuration options and advanced use cases, refer to the official documentation at <https://github.com/kubernetes-sigs/prometheus-adapter/blob/master/docs/config.md>.

With this configuration in place, the metric becomes available through the Kubernetes API, allowing the Horizontal Pod Autoscaler to make scaling decisions based on it.

9.7.3 Scaling workloads horizontally

Now that we have the vLLM metric available in the Kubernetes Custom Metrics API, we can use it to create Horizontal Pod Autoscaling so that the number of vLLM engine replica becomes of a function of the number of requests being handled concurrently.

Listing 9.4 shows a HPA resource that scales vLLM based on the vLLM metric (`vllm_num_requests_waiting`).

Listing 9.4 HPA configuration for vLLM

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: vllm-hpa
  namespace: default
spec:
  scaleTargetRef: #A
    apiVersion: apps/v1
    kind: Deployment
    name: vllm-opt125m-deployment-vllm
  minReplicas: 1 #B
  maxReplicas: 2
  metrics:
  - type: Object
    object:
      metric:
        name: vllm_num_requests_waiting #C
  describedObject:
    apiVersion: v1
    kind: Namespace
    name: default
  target: #D
    type: Value
    value: 1
#A The Deployment to scale. In this case, it's the vLLM engine Deployment we
installed
#B minReplicas and maxReplicas define the lower and upper bounds for scaling
#C the metric exposed via the Prometheus Adapter
#D the target.value == 1 means: "If more than one request is waiting, scale up."
```

HPA (when coupled with a Kubernetes node scaler like Karpenter or Cluster Autoscaler) gives you dynamic elasticity. It automatically scales inference capacity based on live demand, without manual intervention.

Create this HPA resource by running:

```
$ cd ../vllm
$ kubectl apply -f vllm-hpa.yaml
```

With the HPA set up, we can now test it by generating some traffic and observing how the system responds. To simulate a scaling event, we'll flood the vLLM engine with requests and watch the HPA in action.

First, let's create a port-forward to the vLLM engine service so your local machine can send requests into the cluster:

```
$ kubectl port-forward svc/vllm-engine-service 30080:80
```

Leave this running in one terminal window.

In a second terminal, run the traffic generator script. This script uses the OpenAI Python SDK to send a burst of requests to the vLLM engine:

```
$ pip install openai # if not already installed
$ python load-generator.py
```

In a third terminal, watch the HPA scale your Deployment:

```
$ kubectl get pods -w
```

You should see new engine Pods getting created as the number of pending requests rises. You can also observe the metric itself in real-time via the Kubernetes Custom Metrics API:

```
$ kubectl get --raw
/apis/custom.metrics.k8s.io/v1beta1/namespaces/default/metrics/v
llm_num_requests_waiting | jq
{
  "kind": "MetricValueList",
  "apiVersion": "custom.metrics.k8s.io/v1beta1",
  "metadata": {},
  "items": [
    {
      "describedObject": {
        "kind": "Namespace",
        "name": "default",
        "apiVersion": "/v1"
      },
      "metricName": "vllm_num_requests_waiting",
      "timestamp": "2025-05-07T18:21:01Z",
      "value": "37",
      "selector": null
    }
  ]
}
```

As the request backlog builds, the value of this metric should increase, and the HPA will respond accordingly. This little experiment brings the whole monitoring and autoscaling pipeline together: from metric exposure to

Prometheus scraping, Prometheus Adapter publishing, and the HPA scaling in response. You now have a fully dynamic, feedback-driven inference workload on Kubernetes. And just as importantly, when the traffic drops and the queue clears, the HPA will automatically scale down the engine replicas, helping you conserve resources and reduce cost without lifting a finger.

Using this pattern, you can autoscale any application in Kubernetes using a custom metric. And this capability is not limited to metrics available within a Kubernetes cluster either. You can also use external metrics, such as queue length in an external message broker, API rate limits from a third-party service, or even cost signals from a billing system, to drive autoscaling decisions. If you can expose the metric to the Prometheus Adapter (or a similar mechanism), Kubernetes can make scaling decisions based on it. This opens the door to building reactive, data-driven infrastructure that adapts not only to system load but also to real-world business conditions.

What to scale?

Not every application needs to scale, and not every application can scale effectively. Autoscaling is best suited to workloads that are designed for horizontal scaling, when more replicas can be added to handle more load without breaking shared state or coordination.

Classic examples include web applications, APIs, and inference services, all of which handle independent requests that can be split across replicas. These workloads typically scale based on external pressure, like HTTP request volume or pending user queries.

Other applications scale based on queue depth, such as message processors pulling from Kafka, SQS, or RabbitMQ. Here, scaling decisions are driven by internal pressure (how many tasks are waiting to be processed.)

In machine learning, you're mostly dealing with two workload types: batch and online.

Batch jobs (like model training, ETL, or data preprocessing) are usually scheduled with known resource needs. You might request a fixed number of GPUs and just run the job to completion. While the cluster itself may scale up to accommodate the job, the job isn't typically autoscaled midway based on load (at least for now; but this will change in the future). You don't spawn more training workers halfway through an epoch.

Online workloads (like inference) are more dynamic. Traffic can spike unpredictably. That's why inference is a great fit for autoscaling, particularly

when you can expose metrics like queue depth, request latency, or inflight requests.

It's also worth noting that autoscaling isn't just about scaling up. A well-designed HPA setup should scale down too, releasing resources when demand drops, and keeping your cluster lean during quiet hours.

When designing a system for autoscaling, make sure:

- It can handle abrupt replica count changes.
- It avoids sticky session requirements.
- It exposes metrics that accurately reflect load.
- And it's stateless or has minimal local state.

Design your ML systems with these principles in mind, and autoscaling becomes a powerful tool, not just for performance, but for cost control and resilience too.

9.8 Visualizing metrics with Grafana

One of the challenges of modern observability is the sheer volume of data we now have access to. While this data is valuable, it can also create a kind of noise, making it harder to identify what truly matters. Visualization tools are essential for both real-time monitoring and historical analysis, but they still require us to make decisions about which metrics to focus on.

In the past, monitoring was often fragmented and inconsistent across systems. But with the rise of cloud-native infrastructure and Kubernetes, observability has become far more centralized and standardized. Today, we have powerful open source and commercial tools that can aggregate telemetry from across the stack into a single interface.

Grafana is a great example. It's a mature, widely adopted open source project originally built for visualizing time series data, especially Prometheus metrics, and has since grown into a general-purpose observability platform. Grafana connects to dozens of data sources, including Prometheus, Loki, Elasticsearch, PostgreSQL, and even cloud services like AWS CloudWatch and Google Cloud Monitoring.

It solves two important problems. First, it makes it easy to build rich, interactive dashboards that let you slice, filter, and explore metric data from almost any system. Second, it encourages reusability, developers can build and

share dashboards tailored to specific systems, allowing teams to adopt consistent, domain-specific monitoring practices across their infrastructure.

Grafana is now part of the Cloud Native Computing Foundation (CNCF) ecosystem and is backed by a strong developer and user community. It powers monitoring setups at startups, enterprises, and hyperscalers alike.

9.8.1 Exploring Grafana

We already have Grafana in our cluster. It was installed along with Prometheus using kube-prometheus-stack. To access it create a port forward:

```
$ kubectl --namespace monitoring port-forward svc/kube-prometheus-stack-grafana 3000:80
```

Then open <http://localhost:3000> to access the Grafana UI. The default username is “admin”, and the password is “mllops-grafana-pwd”. In Grafana, go to Home → Dashboard and you’ll see a list of dashboards that kube-prometheus-stack preinstalls. Take a moment to explore the dashboards. If this is your first time in Grafana, you’ll definitely find the data in those dashboards intriguing.

When you’re done exploring, go to Home → Dashboards and click on “New” and then “Import”. Next, paste the JSON contents of the file vllm-grafana-dashboard.json and import the dashboard. You’ll now be presented with the dashboard that’s created by the vLLM developer community.

Currently, the dashboard has a host of boring information. That’s because most metrics that make up this dashboard are unavailable. Meaning there’s no data to show. Even though our cluster has one vLLM instance running, most of the boxes are empty since we’re not sending it any requests. To visualize the dashboard when it is actively handling requests you can repeat the steps in section 2.6.3 Scaling Workloads Horizontally.



Figure 9.2 A screenshot of vLLM Grafana Dashboard showing the number of requests and KV cache usage percentage. Grafana dashboards can be shared making it easier for new users to identify a system’s key performance indicators.

Grafana's website (<https://grafana.com/grafana/dashboards>) hosts thousands of these prebuilt dashboards for everything from Kubernetes internals to ML frameworks, databases, CI/CD tools, and even cloud billing metrics. If you're running an open source project in your cluster, chances are someone has already built a dashboard for it.

As your observability footprint grows, having these ready-made dashboards not only saves time, it sets a foundation for building your own visualizations on top of proven metrics. Just remember the goal isn't to display everything, but to surface the right signals for your workload, your team, and your users.

9.8.2 Applying these patterns across your MLOps stack

In this chapter, we focused on monitoring and autoscaling a single component (vLLM) to walk you through the full observability pipeline: metrics exposure, scraping with a ServiceMonitor, exposing those metrics with the Prometheus Adapter, and building actionable dashboards in Grafana (or a similar tool). But the real takeaway isn't just how to monitor vLLM, it's that this pattern applies everywhere.

Whether you're running Airflow, JupyterHub, Ray, Kubeflow, or MLflow, the workflow is the same:

- Ensure the service exposes Prometheus-formatted metrics (/metrics endpoint).
- Create a ServiceMonitor to scrape them.
- Use Prometheus and Grafana to visualize them.
- Use Prometheus Adapter and HPA if you want to scale based on them.

The tools may differ slightly in what metrics they expose, but the architecture stays consistent. You now have the foundational pieces to observe and scale any component in your MLOps platform. Go apply them.

9.9 Emitting and Collecting Traces

Among the three pillars of observability, logs, metrics, and traces, traces are the least commonly implemented, but they can offer the deepest insights when it comes to performance tuning. This is especially true in machine learning inference systems, where a single user request may touch several services: input validation, feature transformation, the model server, and post-processing layers. When your inference workload is deployed as part of a larger microservices architecture, distributed tracing becomes a powerful tool.

While logs and metrics are helpful, they don't always tell you why or where a request is slowing down. Distributed traces let you follow a single request as it flows through the system, connecting the dots between services, identifying bottlenecks, and revealing how long each stage takes. This kind of visibility is particularly useful when performance tuning your model-serving pipeline or trying to debug tail latency issues that appear under high load.

What is a Trace

A trace represents the full lifecycle of a single request or operation as it traverses your infrastructure. Each part of that lifecycle is captured as a *span*, which includes information like the operation name, timestamps, the service involved, and any relevant metadata. For example, a trace might include one span for the router receiving a request, another for the model engine performing inference, and a third for writing results to a cache.

Traces help where metrics cannot. Metrics aggregate over time, but they lose context about individual requests. Logs may capture events, but stitching them together across services is often difficult and compute-intensive. Traces fill this gap by giving you a structured view of a specific invocation, like a call stack that spans your entire infrastructure.

A span contains structured metadata like:

- `trace_id`, `span_id`, and `parent_id`
- `start_time` and `end_time`
- `name` - a short description like `predict()` or `GET /generate`
- `attributes` - contextual data like HTTP status, method, or latency

Once ingested into a backend like Grafana Tempo, you can view traces visually: a waterfall chart of spans showing how time is spent across the system. This makes it much easier to correlate what happened when something goes wrong, or why inference is suddenly slower at peak traffic.

In MLOps systems, developers use tracing for:

- Identifying performance bottlenecks – For example, if traces show that a model server consistently responds in 30ms, but full requires latency is 90ms, you now know to investigate other services in the chain.

- Reduce debugging effort – Instead of combing through logs across multiple applications, a trace lets you reconstruct a requests' journey in seconds
- Dependency mapping – Traces are often used to "stitch" requests and understand the flow of a request through the different components of a distributed system.

9.9.1 Implementing Tracing in ML Workflows

When implementing tracing for ML systems, the most effective approach is to start with the built-in OpenTelemetry integrations that already exist in your technology stack. Most modern ML frameworks and infrastructure tools provide native or community-supported OpenTelemetry instrumentation that can be enabled with minimal configuration. This approach gives you immediate visibility without requiring extensive custom instrumentation. The goal is to get as much automated tracing as possible, because every line of code you have to modify adds complexity, risk, and maintenance overhead. In production environments, the less you have to touch the application logic to observe it, the better.

OpenTelemetry supports two primary approaches to instrumentation: code-based and zero-code. Code-based solutions use official SDKs to generate custom spans within your application logic, giving you fine-grained visibility and full control over how telemetry is emitted. Language specific zero-code instrumentation (<https://opentelemetry.io/docs/concepts/instrumentation/zero-code/>) inject tracing automatically into common libraries and runtimes without any changes to your code. These are ideal when you want observability with minimal friction, or when you're working with systems you can't easily modify. Together, these approaches let you observe both what's happening inside your ML application and at its edges, where it connects to databases, services, and users.

The general pattern involves:

35. Setting up an OpenTelemetry collector to receive trace data (more on this in the next section)
36. Enabling the pre-built instrumentations for each component in your stack (Zero-code instrumentation)
37. Configuring context propagation between components
38. Adding custom spans only where needed to fill visibility gaps (Additional code-based instrumentation)

For example, Ray, PyTorch, FastAPI, and Airflow all offer OpenTelemetry plugins that you can enable with just a few lines of code. This provides the foundation of your tracing infrastructure, which you can then enhance with custom instrumentation for ML-specific operations not covered by standard integrations.

9.9.2 *Collecting Traces*

Traces are typically generated by the inference frameworks' API server (like FastAPI or vLLM) using a tracing SDK, often OpenTelemetry. These spans are exported to a trace collector like Jaeger, Zipkin, Grafana Tempo, or a similar service, which aggregates and stores the trace data for visualization and analysis.

The process typically involves:

- An exporter embedded in your app emitting spans
- A collector aggregating spans and assembling full traces
- A backend (like Grafana Tempo) storing traces and exposing them for search

Most frameworks support this pipeline out of the box, including Ray and vLLM. In fact, Ray natively integrates with OpenTelemetry, which means if you've followed previous chapters, you already have the hooks to emit trace data.

Unlike metrics and logs, which are typically collected exhaustively, traces are sampled, often heavily. Capturing a trace for every request is impractical in high-throughput environments. Instead, systems may sample requests based on a certain criterion.

The goal is to retain the most useful traces, those tied to anomalies, without overwhelming your observability stack. Just be aware that if sampling is not coordinated across services, some spans in a trace may be missing.

9.10 *Handling the dilemma of what to monitor*

Once you have a metrics system in place, the natural question is: *what should we measure?* In practice, this depends on your workload, but there are several well-established frameworks that can help guide you toward meaningful metrics.

Two of the most widely used models are the USE method and the RED method.

- USE (Utilization, Saturation, Errors) is most commonly applied to infrastructure-level monitoring. For each resource (CPU, memory, disk, GPU, network), you should track:

- Utilization – How much of the resource is being used.
 - Saturation – How much demand is waiting (e.g., queue length).
 - Errors – Any faults, retries, or failed operations involving that resource.
- RED (Rate, Errors, Duration) is often applied to services and application endpoints:
 - Rate – The number of requests or operations per second.
 - Errors – The number of failed requests or operations.
 - Duration – The time it takes to complete a request or operation.

These mental models give you a starting point for selecting metrics and creating alerts that capture both system health and user experience.

9.10.1 Monitoring Distributed ML Workloads

For ML systems running on Kubernetes, monitoring becomes more specialized. You'll often deal with distributed training and inference services, both of which introduce unique challenges.

For distributed training, consider:

- Throughput – Samples or batches processed per second.
- Training time per epoch – Useful for identifying performance regressions.
- Resource utilization – Are your workers bottlenecked by CPU, memory, disk, or networking?
- Synchronization delays – In frameworks like PyTorch DDP, stragglers can slow down the whole job.

For inference, the RED method is particularly helpful:

- Request rate – How many inference calls are being made.
- Error rate – Are any requests failing due to timeouts, exceptions, or model errors?
- Latency (duration) – How long it takes to return a prediction.
- Token Throughput - For LLMs, tokens/second/GPU is more revealing than request counts, as it normalizes across varying prompt and generation lengths.

9.10.2 Monitoring GPU Utilization

In most ML workloads, GPUs are the most expensive resource in the system. That's why monitoring GPU usage is essential, not just for performance, but for cost efficiency. You should track:

- GPU memory usage
- GPU compute utilization
- GPU hardware health metrics like temperature, power consumption, and error count (ECC and CUDA errors)
- In distributed training, monitor inter-GPU communication metrics like NVLink bandwidth utilization

This helps you answer critical questions: Are my models using the GPU efficiently? Is there idle capacity? If GPUs are underutilized, it may be time to consider GPU sharing strategies. For basic visibility, start by installing the NVIDIA Device Plugin (<https://github.com/NVIDIA/k8s-device-plugin>), which exposes GPU metrics to Kubernetes. Tools like DCGM (Data Center GPU Manager) (<https://developer.nvidia.com/dcgm>) can provide advanced health checks and alert you before hardware issues impact training runs.

NOTE For deeper GPU analysis, tools like NVIDIA Nsight Systems and PyTorch offer kernel-level profiling that standard metrics miss. While monitoring might show 99% GPU utilization, Nsight can reveal if kernels are actually memory-bound or something else. Teams running periodic profiling against their workloads have found substantial speedups in models that appeared optimal by normal metrics.

9.10.3 Monitoring the Model

While infrastructure monitoring tells you if your system is running, only model monitoring reveals if it's running correctly. ML systems can silently produce increasingly inaccurate results while appearing perfectly healthy from a technical perspective. As explainability and model governance requirements continue to evolve, the field of model monitoring is rapidly advancing with new techniques emerging regularly. The patterns described below represent common approaches, but this is by no means an exhaustive list.

VALIDATION JOB PATTERN

This approach leverages scheduled Kubernetes CronJobs to periodically evaluate model behavior against established baselines. The jobs compare current production data distributions with snapshots captured during training and assess

performance using reference datasets with known ground truth. This pattern offers resource efficiency but provides only periodic visibility into model health. It's particularly effective for detecting both feature drift and concept drift through statistical comparisons, making it well-suited for batch prediction systems where real-time response isn't critical.

REQUEST MIRRORING PATTERN

By duplicating production traffic to monitoring systems without affecting the primary serving path, this pattern enables continuous analysis without risking production performance. It creates shadow copies of incoming requests, allowing for A/B comparison between current and candidate models on identical traffic. This approach preserves production latency while providing continuous monitoring and proves particularly valuable for testing model updates before full deployment. Implementing request mirroring requires careful resource management to avoid contention between production and monitoring workloads.

INLINE MONITORING PATTERN

This more direct approach analyzes production inference in real-time within the serving path itself. By embedding lightweight drift detection directly in the inference pipeline, it captures every request and response for immediate analysis, tracking per-feature statistics and prediction confidence over time. This pattern can detect anomalies at both the individual request level (outliers) and the population level (drift), typically implemented through sidecars or as model server middleware. For this approach to be viable, monitoring components must be carefully optimized to minimize latency impact on predictions.

LOG-BASED ANALYSIS PATTERN

By post-processing structured logs from model servers, this pattern enables comprehensive drift detection with minimal production impact. It records enriched request/response logs containing key features and prediction details, then performs statistical analysis asynchronously on these log streams. This approach leverages existing log infrastructure (like ELK Stack or Grafana Loki) enhanced with custom processors for ML-specific analysis. Beyond basic drift detection, log-based analysis enables historical trend identification across model versions and cohort analysis.

As model governance requirements continue to evolve and machine learning systems become more deeply integrated into critical business processes, we can expect monitoring approaches to become more sophisticated. Emerging

techniques are beginning to appear in production systems to make observability, explainability and governance more robust.

As regulatory frameworks and industry-specific compliance requirements mature, these capabilities will shift from optional enhancements to essential components of production ML systems. Organizations that invest in comprehensive model monitoring now will be better positioned to adapt to these evolving requirements while maintaining the performance and reliability of their ML applications.

Summary

- Logs, metrics, and traces work together to provide a complete picture of system behavior, with each offering unique insights into different aspects of your ML workloads.
- OpenTelemetry provides a unified approach to instrumentation across your stack, future-proofing your observability strategy regardless of which backends you choose.
- Prometheus' widespread adoption in the Kubernetes ecosystem means virtually every ML tool already supports it, providing a consistent metrics collection pipeline across your platform.
- Use the Prometheus Adapter to expose ML-specific metrics through the Kubernetes API, enabling intelligent scaling decisions.
- Grafana makes monitoring both off-the-shelf and custom software straightforward. Community-created dashboards significantly reduce the effort needed to visualize complex ML metrics, providing immediate visibility without extensive configuration.
- The USE method (Utilization, Saturation, Errors) provides a framework for infrastructure monitoring, while the RED method (Rate, Errors, Duration) offers clarity for service-level observability. Together, they create a comprehensive monitoring strategy that covers both system resources and user experience.
- Implement regular validation jobs or continuous monitoring to detect model and data drift before they impact business outcomes.
- Tracing connects the dots across system boundaries and provides the critical context to pinpoint bottlenecks, map dependencies between services, and understand request flow.
- Resource utilization patterns for ML workloads differ significantly from traditional applications, requiring specialized metrics for GPUs and other accelerators.

Appendix

The appendix explains the steps for setting up the foundational AWS infrastructure required to create a Kubernetes cluster using Amazon Elastic Kubernetes Service (Amazon EKS) cluster using Terraform.

Creating the base infrastructure using Terraform

This section contains steps for provisioning the base infrastructure required by the MLOps components discussed in this book.

To provision the infrastructure, you'll need to have an AWS account, AWS API credentials configured, and AWS CLI installed on your local machine. Please see AWS documentation for configuring credentials (<https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html>).

Once you've installed AWS CLI and configured, you can run this command to check if it's functioning properly:

```
$ aws sts get-caller-identity
Account: XXXXXXXXXXXX
Arn: arn:aws:iam::XXXXXXXXXXXX:user/realvarez
UserId: AIDAJHI4KRXAY6NXEEEUO
```

The IAM role or user you use for this book will need permissions to create resources like Amazon EC2 instances, storage volumes, and a Kubernetes cluster. To keep things simple, we recommend attaching the Power User policy (<https://docs.aws.amazon.com/aws-managed-policy/latest/reference/PowerUserAccess.html>) to your role or user. This practice is acceptable in a lab environment. Such permissive policies shouldn't be used in production. Please see the following document to learn best practices about securing IAM: <https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html>

Amazon Elastic Kubernetes Service (Amazon EKS) is a managed Kubernetes service to run Kubernetes in the AWS cloud and on-premises. EKS provides a managed Kubernetes control plane that simplifies provisioning and maintaining clusters. You can use a variety of tools such as the AWS Management Console, AWS CLI, infrastructure-as-code tools (CloudFormation, Terraform, Pulumi, and CDK). The easiest way to create a cluster is using eksctl, which is a command line tool to create EKS clusters and associated infrastructure such as VPCs, subnets, nodes.

NOTE AWS provides Terraform (<https://github.com/aws-ia/terraform-aws-eks-blueprints>) and CDK templates (<https://github.com/aws-quickstart/cdk-eks-blueprints>) to create clusters.

We'll use *Terraform* (<https://www.terraform.io>), which is an open-source infrastructure as code (IaC) tool that allows you to define and provision infrastructure resources across multiple cloud providers using a simple, declarative language. Terraform supports a wide range of cloud providers, including AWS, Azure, Google Cloud, and many others, allowing you to manage infrastructure resources in a cloud-agnostic manner.

Please install Terraform on your local machine. You can find the instructions to install Terraform in its documentation (<https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli>).

Clone the book's code repository and go to the "Chapter 3" directory:

```
$ git clone https://github.com/mlops-on-kubernetes/Book.git  
$ cd 'Book/Chapter 3/eks'
```

In this directory, you'll notice main.tf Terraform file. This file includes all the AWS resources we need to run this MLOps cluster. It is a best practice to modularize Terraform code, which means breaking it down into smaller .tf files that can be changed independently. For illustration purposes, we've ignored this practice and declared all resources within the same file (main.tf).

main.tf explained

This file contains the foundational code that sets up the environment for deploying the Amazon EKS-based Kubernetes cluster. It contains seven sections that control the configuration for:

1. The Terraform environment

2. VPC
3. EKS cluster
4. EBS storage class
5. EFS filesystem

Most Terraform project start by defining versions for Terraform itself and any supporting modules. In Terraform, a *provider* is a plugin that acts as an interface between Terraform and an external service, such as a cloud provider. Providers enable Terraform to manage resources, interact with APIs, and perform operations across various technologies without requiring users to learn the intricacies of each individual API. They encapsulate the logic needed for authentication, resource creation, updating, and deletion within the target environment. There are two providers configured in main.tf. Terraform uses the "hashicorp/aws" (<https://registry.terraform.io/providers/hashicorp/aws>) for provisioning AWS resources, such as an EKS cluster or VPC.

Similarly, Terraform uses "hashicorp/helm" (<https://registry.terraform.io/providers/hashicorp/helm/latest>) to manage the lifecycle of resources deployed inside the Kubernetes cluster.

Let's walkthrough main.tf line by line to understand the Terraform code and the resources it deploys. Listing 10.1 explains the first section, the Terraform block, in main.tf.

Listing 10.1 The terraform block in main.tf

```
terraform {
  required_version = ">= 1.3.2" #A
  required_providers {
    aws = {
      source  = "hashicorp/aws" #B
      version = ">= 5.83"
    }
    helm = {
      source  = "hashicorp/helm" #C
      version = ">= 2.9"
    }
  }
}

#A The minimum Terraform version required to run this code (1.3 or later)
#B The provider used for deploying any AWS resources
#C The Helm provider, used for deploying applications onto Kubernetes.
```

Listing 10.2 shows the AWS provider configuration. Here we just define the AWS region in which we want to provision resources.

Listing 10.2 The AWS provider block in main.tf

```

provider "aws" {
    region = local.region #A
}
#A The region is dynamically set using a local variable (defined later)

```

Next comes the Kubernetes block shown in Listing 10.3. This block configures the Kubernetes provider, which allows Terraform to manage Kubernetes resources.

Notice that it retrieves the value for the Kubernetes cluster endpoint using `module.eks.cluster_endpoint`. A *module* in Terraform is a collection of Terraform configuration files that are used together. Modules are used to package and reuse resource configurations. The “eks” module is defined in the `eks.tf` file.

Listing 10.3 The Kubernetes provider block in main.tf

```

provider "kubernetes" {
    host           = module.eks.cluster_endpoint #A
    cluster_ca_certificate =
    base64decode(module.eks.cluster_certificate_authority_data) #B

    exec {
        api_version = "client.authentication.k8s.io/v1beta1"
        command     = "aws"
        args        = ["eks", "get-token", "--cluster-name",
        module.eks.cluster_name]
    } #C
}
#A Retrieves the Kubernetes API server endpoint from the “eks” module
#B Retrieves the certificate authority data for the Kubernetes cluster
#C Configures how Terraform authenticates to the Kubernetes cluster using the aws eks get-token command

```

Next comes the Helm provider block, which you’ll notice looks almost identical to the Kubernetes provider block. You can think of Helm as the package manager for Kubernetes. It simplifies the process of deploying and managing applications on Kubernetes clusters by using a packaging format called Helm charts.

The Helm provider for Terraform allows you to deploy and manage Helm charts in Kubernetes clusters using Terraform. The provider is the interface between Terraform and Helm that enables users to install, upgrade, and delete Helm charts as part of Terraform-managed infrastructure.

In essence, we’re saying that after Terraform creates the EKS-based Kubernetes cluster, it should install some Helm charts. At this stage, we’re not specifying which charts to install; we’re just configuring the Helm provider in Terraform. This configuration sets up the groundwork for later defining and managing Helm releases within our Terraform code. It prepares Terraform and

Helm for deploying applications via Helm charts once the cluster is up and running.

Listing 10.4 The Helm provider block in main.tf

```
provider "helm" {
  kubernetes {
    host           = module.eks.cluster_endpoint
    cluster_ca_certificate =
    base64decode(module.eks.cluster_certificate_authority_data)

    exec {
      api_version = "client.authentication.k8s.io/v1beta1"
      command     = "aws"
      args        = ["eks", "get-token", "--cluster-name",
                    module.eks.cluster_name]
    }
  }
}
```

Lastly, main.tf defines data sources and variables that can be used throughout your Terraform configuration.

For instance, suppose you need to get the Amazon Resource Number (ARN), a unique identifier, for an S3 bucket. You can do so in Terraform by using a data block like this:

```
data "aws_s3_bucket" "example" {
  bucket = "my-existing-bucket"
}
```

You can then reference the ARN using `data.aws_s3_bucket.example.arn` elsewhere in your configuration.

Listing 10.5 show the remainder of main.tf. This section includes defining variables to be used later. For example, the `aws_availability_zones` is a variable that stores the list of the Availability Zones (AZs) in the current AWS Region. Similarly, We specify that Terraform should deploy this infrastructure in the "us-west-2" (Oregon) AWS Region.

Listing 10.5 Data and variables section of main.tf

```
data "aws_availability_zones" "available" {} #A

locals {
  name   = "machine-learning" #B
  region = "us-west-2" #C

  vpc_cidr = "10.0.0.0/16" #D
  azs      = slice(data.aws_availability_zones.available.names,
                  0, 3) #E
```

```

tags = {
    EKSCluster  = local.name #F
}
}

#A Retrieve and store the list of Availability Zones in a variable called
aws_availability_zones
#B Define a local variable 'name' with the value "machine-learning"
#C Set the AWS region to "us-west-2" (Oregon) for this configuration
#D Specify the CIDR block for the VPC as "10.0.0.0/16", which provides a range of IP
addresses for the network
#E Create a list of the first 3 Availability Zones from the data source
#F Define a tag with key "EKSCluster" and value equal to the 'name' variable, used for
resource labeling and organization

```

Now that we've setup Terraform and configured providers in main.tf, the next step is to create the Kubernetes cluster. But there's some prework required before we can create a cluster. First, we must create a new Virtual Private Cloud and setup networking. This includes carving the VPC up into multiple subnets.

Creating VPC for an EKS cluster

A VPC is a logically isolated virtual network in the AWS cloud that allows you to launch and manage resources securely. This VPC is then further divided into multiple subnets. In main.tf, we define a VPC that has two subnets in every Availability Zone: one public subnet with direct internet access and another private subnet that has access to the internet through a NAT Gateway.

Listing 10.6 explains `vpc.tf`. You'll notice a new term here: module. A module in Terraform is a self-contained package of Terraform configurations that manages a set of related resources as a group. It allows you to create reusable and shareable components of infrastructure code, which usually improves consistency, scalability, and efficiency in Terraform projects. Modules can be called from other configurations, enabling you to organize complex infrastructure into manageable pieces and reuse common patterns across different projects or environments.

The Terraform Registry hosts an extensive collection of publicly accessible Terraform modules, meticulously crafted to configure diverse types of common infrastructure. These modules, curated by industry experts and available at no cost, significantly streamline the process of resource provisioning by offering sensible default configurations. By leveraging modules, users can substantially reduce their learning curve and expedite infrastructure deployment.

Modules abstract away complex configurations, allowing users to focus on essential parameters while relying on pre-defined defaults for the rest. For instance, when creating a VPC, users need not be intimately familiar with every

configuration option. Instead, they can simply specify the values they wish to customize, and the module applies default values for the remaining parameters.

The code uses the "terraform-aws-modules/vpc/aws" module to create a VPC. Within the module block, we can then tweak configuration parameters such as name, region, and other properties of the VPC.

Listing 10.6 VPC configuration in main.tf

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 5.0"

  name = local.name
  cidr = local.vpc_cidr

  azs           = local.azs
  private_subnets = [for k, v in local.azs :
    cidrsubnet(local.vpc_cidr, 4, k)] #A
  public_subnets = [for k, v in local.azs :
    cidrsubnet(local.vpc_cidr, 8, k + 48)] #B

  enable_nat_gateway = true #C
  single_nat_gateway = true #D

  public_subnet_tags = {
    "kubernetes.io/role/elb" = 1 #E
  }

  private_subnet_tags = {
    "kubernetes.io/role/internal-elb" = 1 #F
    "karpenter.sh/discovery" = local.name #G
  }

  tags = local.tags
}

#A Create private subnets by using a for loop to iterate over the Availability Zones
and applying the cidrsubnet function to calculate subnet CIDRs. It adds 4 bits to the
VPC CIDR, allowing for up to 16 private subnets.
#B Similarly create public subnets but add 8 bits to the VPC CIDR and offsets by 48,
ensuring public subnets have different address ranges from private ones. This allows
for up to 256 public subnets.
#C Enable the creation of a NAT Gateway, which allows instances in private subnets
to access the internet while remaining private.
#D Configure a single NAT Gateway for all Availability Zones, which is more cost-
effective but less fault-tolerant than having one per AZ.
#E Add a tag that is used by the AWS Load Balancer Controller for creating internet-
facing load balancers
#F Tag private subnets for internal-facing load balancers
#G Add a tag to private subnets for Karpenter to discover and use these subnets for
launching new worker nodes.
```

After defining the VPC, we can now create an EKS cluster.

Creating an EKS cluster

Just like the code used "terraform-aws-modules/vpc/aws" to create a VPC, it uses "terraform-aws-modules/eks/aws" to create an EKS cluster. There's not a lot we modify here. In listing 10.7, you'll see the cluster configuration.

The most important configuration is cluster_compute_config, enabling which enables EKS Auto Mode. EKS Auto mode a feature that automates critical tasks such as Kubernetes worker node provisioning, scaling, security patching, and component updates. By integrating services like Karpenter for autoscaling, AWS Load Balancer Controller for networking, and EBS CSI driver for storage, EKS Auto Mode offers a simplified, efficient, and secure Kubernetes experience that optimizes costs and enhances application availability.

We've enabled Auto Mode to simplify cluster provisioning. You may choose not to do this in your cluster if you're familiar with managing Kubernetes cluster. We'll leave that decision up to you, just know that this cluster has Auto Mode enabled.

Listing 10.7 EKS cluster configuration

```
module "eks" {
  source  = "terraform-aws-modules/eks/aws" #A
  version = "~> 20.31" #B

  cluster_name          = local.name #C
  cluster_version       = local.cluster_version #D

  enable_cluster_creator_admin_permissions = true #E

  cluster_compute_config = {
    enabled      = true #F
    node_pools   = ["general-purpose"] #G
  }

  cluster-addons = {
    aws-efs-csi-driver     = {
      service_account_role_arn =
      module.efs_csi_driver_irsa.iam_role_arn #H
    }
  }

  enable_efa_support = true #I
  enable_irsa = true #J

  vpc_id      = module.vpc.vpc_id #K
  subnet_ids = module.vpc.private_subnets #L

  tags = local.tags
}

#A Specifies the source of the EKS module from the Terraform Registry
#B Sets the module version, using a tilde to allow patch updates but not minor or
major version changes
#C Sets the EKS cluster name using a local variable
#D Defines the Kubernetes version for the cluster, also using a local variable
#E Grants admin permissions to the IAM entity creating the cluster
#F Enables EKS Auto Mode
#G Specifies the node pool to be created in Auto Mode
#H Role for the EFS CSI Driver
#I Enables support for Elastic Fabric Adapter, a network interface for high-
performance computing
#J Enables IAM Roles for Service Accounts, allowing fine-grained access control for
pod-level permissions.
#K Specifies the VPC ID for the EKS cluster
#L Defines the private subnets for the EKS cluster
```

Next, we must create storage resources so workloads can persist data.

Provisioning storage for workloads

Workloads such as JupyterHub and Keycloak require storage to persist data. The type of storage you require depends on the use case. For instance, when an application requires high throughput, low latency storage, block storage is the preferred storage system. In other cases, you may need large amounts of scalable, cost-effective storage for unstructured data, which is where object storage excels.

To simplify things, we'll only use two types of storage in this book. We will use Amazon EBS to provide storage for any workload that requires persistent storage. Whenever we require shared storage or the ability to read and write from multiple systems simultaneously, we'll use an Amazon EFS-based NFS filesystem.

EKS Auto Mode automates EBS volume provisioning. All we must do is create a Storage Class as shown in listing 10.8.

Listing 10.8 Amazon EBS Storage Class gp3

```
resource "kubernetes_storage_class" "ebs-gp3-sc" {
  metadata {
    name = "gp3" #A
  }

  storage_provisioner = "ebs.csi.eks.amazonaws.com" #B
  reclaim_policy     = "Delete" #C
}

#A Name of the storage class
#B Storage provisioner as required by the EKS cluster
#C Deletes volumes when its associated PersistentVolumeClaim is deleted
```

Amazon EFS configuration is next. First, we must install the addon so the EKS cluster can interact with the Amazon EFS service. We didn't have to perform this step for Amazon EBS because EKS Auto Mode automates that step.

In listing 10.9, we use IAM Roles for Service accounts Terraform module to create an IAM role that the EFS CSI Driver requires.

Listing 10.9 Amazon EFS CSI Driver IAM role

```
module "efs_csi_driver_irsa" {
  source      = "terraform-aws-
modules/iam/aws//modules/iam-role-for-service-accounts-eks"
  role_name   = "${local.name}-efs-csi-driver" #A
  role_policy_arns = {
    policy = "arn:aws:iam::aws:policy/service-
role/AmazonEFSCSIDriverPolicy" #B
  }
  oidc_providers = {
```

```

    main = {
        provider_arn           = module.eks.oidc_provider_arn
        namespace_service_accounts = ["kube-system:efs-csi-
controller-sa"]
    }
}
tags = local.tags
}

#A Name of the IAM Role the CSI driver can assume
#B Policy attached to the role

```

With the Amazon EFS CSI Driver installed in the cluster, we can use Kubernetes API to create an Amazon EFS file system.

```

resource "aws_efs_file_system" "efs-share" {
    creation_token = "ml-share"
    encrypted      = true
    tags = {
        Name = "ml-share"
    }
}

```

Following Amazon EFS best practices, we've also created an EFS Access Point to enhance file system security capabilities. These can be especially useful in a multitenant environment where multiple tenants share the same underlying file system but are still isolated. EFS *Access Points* create isolated entry points to an EFS file system, allowing for fine-grained control over how applications interact with shared storage. In the configuration shown in listing 10.10, the Access Point creates a root directory at "/shared" with full permissions (0777) to UID 100.

Listing 10.10 Amazon EFS Access Point

```

resource "aws_efs_access_point" "efs-ap" {
    file_system_id = aws_efs_file_system.efs-share.id

    posix_user {
        gid = 100
        uid = 1000
    }

    root_directory {

        creation_info {
            owner_uid    = 1000
            owner_gid    = 100
            permissions = "0777"
        }
        path = "/shared"
    }
}

```

To ensure optimal performance and accessibility of our EFS file system, we've implemented EFS mount targets across our VPC's private subnets. This configuration, as shown in Listing 10.11, creates mount targets in each private subnet. This approach follows AWS best practices for high availability and fault tolerance, ensuring that EC2 instances in any private subnet can access the EFS file system with low latency. The security group attached to the mount target is set to permit traffic that's local to the VPC.

Listing 10.11 Amazon EFS Mount Target

```
resource "aws_efs_mount_target" "efs-share" {
  file_system_id = aws_efs_file_system.efs-share.id
  for_each       = toset(module.vpc.private_subnets)
  subnet_id      = each.value
  security_groups = [aws_security_group.efs.id]
}

resource "aws_security_group" "efs" {
  name          = "${local.name}-efs"
  description   = "Allow inbound NFS traffic from private subnets
of the VPC"
  vpc_id        = module.vpc.vpc_id

  ingress {
    description = "Allow NFS 2049/tcp"
    cidr_blocks = [local.vpc_cidr]
    from_port   = 2049
    to_port     = 2049
    protocol    = "tcp"
  }

  tags = local.tags
}
```

Next, we setup a storage class for EFS:

```
resource "kubernetes_storage_class" "efs" {
  metadata {
    name = "efs-sc"
  }

  storage_provisioner = "efs.csi.aws.com"
  reclaim_policy     = "Retain"
  parameters = {
  }
}
```

Creating a Shared Storage using Amazon EFS

In this book, we use Amazon EFS to share files on a temporary basis. For example, imagine data scientists on our team needed a way to share Jupyter

Notebooks or other data. In this setup, they'll use the Amazon EFS filesystem for these purposes.

First, we'll create a namespace inside the Kubernetes cluster and call it Jupyter. Next, we'll create a PersistentVolumeClaim(PVC) and a PersistentVolume (PV). When we want to mount this filesystem into Pods, we'll add a reference to this PVC. Any such Pod will have access to read and write to this filesystem.

Since we've already created an EFS filesystem, we must create a PV and reference the file system id. This PV will get mounted into a Pod it has the associated PVC.

Listing 10.12 Amazon EFS PersistentVolume

```
resource "kubernetes_persistent_volume_claim" "efs-pvc" {
  metadata {
    name = "efs-claim" #A
    namespace = "jupyter" #B
  }
  spec {
    access_modes = ["ReadWriteMany"]
    resources {
      requests = {
        storage = "5Gi" #C
      }
    }
    storage_class_name = "efs-sc" #D
  }
}

resource "kubernetes_persistent_volume" "jupyter-efs-pv" {
  metadata {
    name = "efs-pv" #E
  }
  spec {
    storage_class_name = "efs-sc"
    capacity = {
      storage = "5Gi"
    }
    access_modes = ["ReadWriteMany"]
    persistent_volume_source {
      csi {
        driver      = "efs.csi.aws.com"
        volume_handle = format("%s::%s",
aws_efs_file_system.efs-share.id, aws_efs_access_point.efs-
ap.id) #F
        read_only      = false
      }
    }
  }
}
#A Name of the PVC
#B The PVC should be created in "jupyter" namespace within
Kubernetes cluster
#C This value is ignore in Amazon EFS
#D Name of the storage class
#E Name of the PV
#F Amazon EFS Access Point
```

Infrastructure costs

When you create an EKS cluster, you get a Kubernetes cluster whose control plane is managed for you. This control plane is designed to be as reliable and

resilient as possible. It better be since you pay roughly \$0.10 per hour for an EKS cluster. If you plan to deploy the architecture described in the book, you'll be paying for an EKS cluster and other resources you keep running in the cluster. The cluster itself costs roughly \$72 a month in the US West (Oregon) AWS region.

Besides EKS, you will also pay for storage, worker nodes, load balancers, and networking. It's difficult to give you an exact dollar amount you'd spend on building the infrastructure discussed in this book as it depends on the duration for which you keep resources running. If you keep the infrastructure running for a month, you can expect to spend between \$50 and \$100 for worker nodes.

At various stages in this book, you'll find instructions to delete the infrastructure we deploy. To keep costs low, we recommend cleaning these resources. If you create any data or create a resource, be sure to delete it to avoid any charges. AWS does offer a free tier for many of its services, but it's important to note that EKS is not included in the free tier. However, some of the underlying resources you might use with EKS, such as EC2 instances for worker nodes, may be eligible for free tier benefits depending on your account status and usage. We recommend taking advantage of AWS Cost Explorer and AWS Budgets to monitor and control your spending.

To avoid paying for EKS, you can also use any other Kubernetes cluster. However, please know that the configuration instructions and helper scripts included with this book are designed for AWS, specifically Amazon EKS. You can change these scripts to match your environment if you aren't on AWS. If you do so, consider sharing them by sending us a pull request on this book's GitHub repository. Thanks in advance!

Deploying the template

Deploying the resources is a straightforward process. The key is to have (virtually) unlimited access to the AWS account. You may run into two issues:

- Permissions – You may not have permissions to create certain resources in your account. This can be especially if someone else manages your environment. It will likely not occur on your personal AWS account.
- Quotas – You may not be able to create AWS resources because the default limits are too restrictive in your case. You are more likely to run into this if you have a new AWS account. If you run into a quota or a limit issue, you can request a quota increase as described here:
<https://docs.aws.amazon.com/servicequotas/latest/userguide/request-quota-increase.html>

To deploy the stack you must install Terraform by following the instructions listed in Terraform documentation (<https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli>).

After installing Terraform, initialize it by running init in the “eks” directory:

```
$ terraform init
```

Deploy resources by running:

```
$ terraform apply
```

It takes about ten minutes to create the cluster. Once the cluster is ready, run the following command to setup kubectl:

```
$ aws eks update-kubeconfig --name mlops-cluster --region us-west-2
```

With kubeconfig updated, you can interact with the Kubernetes cluster using kubectl. Let’s test it by running:

```
$ kubectl get namespaces
NAME        STATUS   AGE
default     Active   1h
jupyter     Active   1m
kube-node-lease Active  1h
kube-public  Active   1h
kube-system Active   1h
```

If you see a similar output, you’re good to go. If not, you must troubleshoot what went wrong. Usually, it’s a permission error or quota issue with the AWS account or user.

Hopefully, you didn’t face any issues when creating this cluster. If so, congratulations! You have the base infrastructure needed to begin building an MLOps system.

Acknowledgements

As I reflect on the journey of writing this book, I am filled with gratitude for all those who have supported me along the way. This book represents not just my technical knowledge, but also the culmination of countless interactions, learning experiences, and support from numerous individuals who have contributed to its creation.

First and foremost, I want to express my deepest gratitude to my family - my loving wife Nila, and my wonderful children Shivani and Theju. Their unwavering support and understanding as I dedicated countless evenings and weekends to this project has been the foundation of my success. Nila's patience during long writing sessions, her encouragement during moments of doubt, and her willingness to take on extra responsibilities to give me the time and space to work have been invaluable. My children's understanding when I missed family time and their bright smiles that kept me going have meant the world to me. This book is as much a testament to their love and support as it is to my efforts.

Special thanks to my colleagues at Amazon who provided invaluable guidance and reviewed my approaches and architectures throughout this book. Their expertise in cloud computing, machine learning, and Kubernetes has been instrumental in refining my ideas and ensuring the accuracy and relevance of my content. The time they took from their busy schedules to offer insights, challenge my assumptions, and share real-world experiences has significantly enhanced the quality and practicality of this book. Their contributions have made this work not just mine, but a collaborative effort reflecting the collective knowledge of our Kubernetes and Machine Learning community.

I am indebted to my friends who served as my steadfast cheerleaders, offering constant encouragement and motivation throughout this journey. Their genuine interest in my progress, their willingness to listen to my ideas (even when they were highly technical), and words of encouragement during challenging times kept me going. They reminded me of the importance of my work when I was too close to see it, and celebrated my milestones, no matter how small. Their friendship and support have been a source of strength and inspiration throughout this process.

My gratitude extends to the vibrant open-source communities, particularly Ray, MLFlow, Spark, Kubeflow and others, whose innovations have inspired much of my work. The spirit of collaboration and knowledge-sharing in these communities has been a guiding light for me. I've built upon the foundations they have laid, and I hope this book contributes back to the collective knowledge pool.

I also acknowledge the authors whose books on related topics have influenced my perspective and approach. Their works have been my teachers, challenging me to think deeper and strive for clarity in my explanations. The insights I've gained from their writings have shaped my understanding of the field and influenced how I've chosen to present complex topics.

Finally, I want to recognize the wealth of knowledge shared through various online platforms including Medium, GitHub, and Reddit. These resources have been an invaluable source of cutting-edge information, practical tips, and real-world experiences. The discussions, code snippets, and articles I found on these platforms helped me refine my technical demonstrations and elevate the overall quality of my work. The global community of developers and data scientists who freely share their knowledge have been silent contributors to this book, and I am deeply appreciative of their collective wisdom.

- Elamaran

I'd like to thank my family for letting me write this book. Without them nothing will be possible.

At AWS, I'd like to thank so many colleagues, including, Jafar Shameem, Rohit Arora, Anish Kantawala, Michael Hausenblas, Apoorva Kulkarni, Vara Bonthu, Florian Stahl, Simon Reichert and all the contributors to the Data/AI on EKS project.

- Re Alvarez Parmar

About the authors

Eelamaran (Ela) Shanmugam is a Sr. Specialist Solutions Architect with Amazon Web Services with over 20 years of experience in architecting, developing and day 2 operations of large scale enterprise systems and applications. Ela helps AWS customers and partners to build products and services using modern technologies to enable their business. Ela is a Container, App Modernization, Observability, Generative AI and Machine Learning SME and helps AWS partners and customers design and build scalable, secure, and optimized workloads on AWS. Being a Sr. technologist, Ela's focus is on modern application development, cloud migration, modernization and automation. Being at AWS, Ela enjoys contributing to open source, public speaking, mentoring, and publishing engaging technical content such as AWS Whitepapers, AWS Blogs, Internal articles. Ela is based out of Tampa, Florida, and you can reach Ela on Twitter @IamElaShan and on GitHub.

Re Alvarez Parmar has spent over two decades architecting, building, and operating enterprise systems and infrastructure. In his current role at AWS as Principal Solutions Architect, he helps some of most the sophisticated Automotive companies modernize and scale their software architecture using cloud. His focus over the past few years has been cloud-native architecture, modern application development, and distributed systems.