O'REILLY

Generative AI on Kubernetes

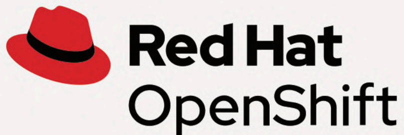
Operationalizing Large Language Models

Early Release

RAW & UNEDITED

Compliments of

Roland Huß
& Daniele Zonca



The application platform for innovation

To meet the constant demand and realize their boldest ideas, teams need a comprehensive and consistent open hybrid cloud platform they can trust: Red Hat OpenShift

Learn more

Generative AI on Kubernetes

Operationalizing Large Language Models

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Roland Huß and Daniele Zonca

O'REILLY®

Generative AI on Kubernetes

by Roland Huss and Daniele Zonca

Copyright © 2026 Roland Huss and Daniele Zonca. All rights reserved.

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

Editors: Angela Rufino and John Devins
Production Editor: Katherine Tozer
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

April 2026: First Edition

Revision History for the Early Release

2025-03-12: First Release
2025-07-01: Second Release
2025-08-07: Third Release
2025-10-17: Fourth Release
2025-11-21: Fifth Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098171926* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Generative AI on Kubernetes*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author(s) and do not represent the publisher's views. While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-17186-5

[FILL IN]

Table of Contents

Brief Table of Contents (*Not Yet Final*) ix

1. Introduction 11
- Challenges running Generative AI at scale 13
- Kubernetes for AI Workloads 15
- DevOps and MLOps 16
- Overview 19
  - Inference 19
  - Training 20

2. Deploying Models 23
- "It works on my machine" 24
- Model Server 26
  - vLLM 29
  - Hugging Face Text Generation Inference 31
  - Other model servers 32
- Model Server Controller 34
  - DIY - Do It Yourself 35
  - KServe 38
  - Ray Serve and KubeRay 42
- Lessons learned 46

3. Model Data 47
- Model Data Storage Formats 48
  - Weight-Only Formats 49
  - Self-contained Formats 51
  - ONNX 54
  - Safetensors 55

GGUF and GGML 57
What’s next ? 59
Model registry 60
Hugging Face Model Hub 61
MLflow Model Registry 63
Kubeflow Model Registry 66
OCI Registry 70
Accessing model data in Kubernetes 71
OCI image for storing model data 74
Modelcars 77
OCI Image Volume Mounts 84
More Information 85

4. Model Observability. 87
Understanding LLM 88
Prefill 91
Decode 94
Observability stack and configuration 96
Logs 96
Metrics 98
Tracing 100
Model Server Metrics 102
Time To First Token (TTFT) 103
Time Per Output Token (TPOT) or Inter Token Latency (ITL) 103
Throughput 103
Latency 104
Other metrics 104
GPU usage Monitoring 106
Quality Metrics 106
Responsible AI 108
Explainability 108
Fairness 109
Model Safety: Hallucination and Guardrails 109
Lessons learned 113

5. Kubernetes and GPUs. 115
GPU Discovery 117
Node Feature Discovery 117
GPU Feature Discovery 119
Kubernetes GPU Device Plugins 120
GPU Workload Scheduling 122
Label-based scheduling 122

Resource-based scheduling 126
Dynamic Resource Allocation 127
NVIDIA GPU Operator 129
Operator configuraton with ClusterPolicy 131
Sub-GPU Allocation 132
Multi-GPU Inference 137
Throughput Scaling 138
Model Parallelism 140
Single-Node versus Multi-Node inference 143
GPU Resource Optimizations 147
Lessons learned 149

6. Running In Production. 151
Model and runtime tuning 152
Language model compression 155
Model performance benchmark 157
vLLM runtime parameters tuning 159
Autoscaling 164
Optimize vLLM startup time 167
LLM-aware routing 170
Disaggregated Serving 177
Lessons learned 181

7. Model Customization. 183
Introduction to LLM creation 183
Tuning a Model 188
Running Tuning Jobs on Kubernetes 194
Kubeflow Trainer 196
Other frameworks 203
Lessons learned 206

8. AI-Driven Applications. 207
Architectural Patterns 208
Kubernetes Workload Types 208
Chat Applications 209
Backend AI Services 212
Retrieval-Augmented Generation 217
RAG Components 219
Document Ingestion 221
User Query Processing 223
RAG on Kubernetes 226
Agentic Workflows 229

Agentic Frameworks and Runtimes 233
OpenAI's Responses API 234
Agents on Kubernetes 235
Multi-Agent Systems 238
Ambient Agents 241
Lessons Learned 242

Brief Table of Contents (*Not Yet Final*)

Chapter 1: Introduction (available)

Chapter 2: Deploying Models (available)

Chapter 3: Model Data (available)

Chapter 4: Model Observability (available)

Chapter 5: Kubernetes and GPUs (available)

Chapter 6: Running In Production (available)

Chapter 7: Model Customization (available)

Chapter 8: Job Scheduling Optimization (unavailable)

Chapter 9: AI-Driven Applications (available)

Chapter 10: Running Agentic Applications in Production (unavailable)

CHAPTER 1
Introduction

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

The release of ChatGPT in 2022 was a watershed moment for the IT world. Overnight, it seemed like everything changed—not because of entirely new concepts, but due to the exponential growth in both model parameters and the sheer volume of training data. This explosion of data and model complexity propelled AI into new territory, with capabilities that were previously unimaginable.

In the world of physics, we describe such moments as phase transitions - when small, gradual changes suddenly lead to dramatic shifts in behavior. The rise of large language models (LLMs) mirrors this idea. For years, AI had been steadily evolving, but the leap in model size, compute power, and data availability pushed it beyond a tipping point. These models began exhibiting human-like text generation and comprehension, disrupting entire industries and resetting our expectations of what AI can do. The graph in Figure 1-1 shows the explosive growth of these parameters and the vast data sources that have driven AI's evolution over the past few years.

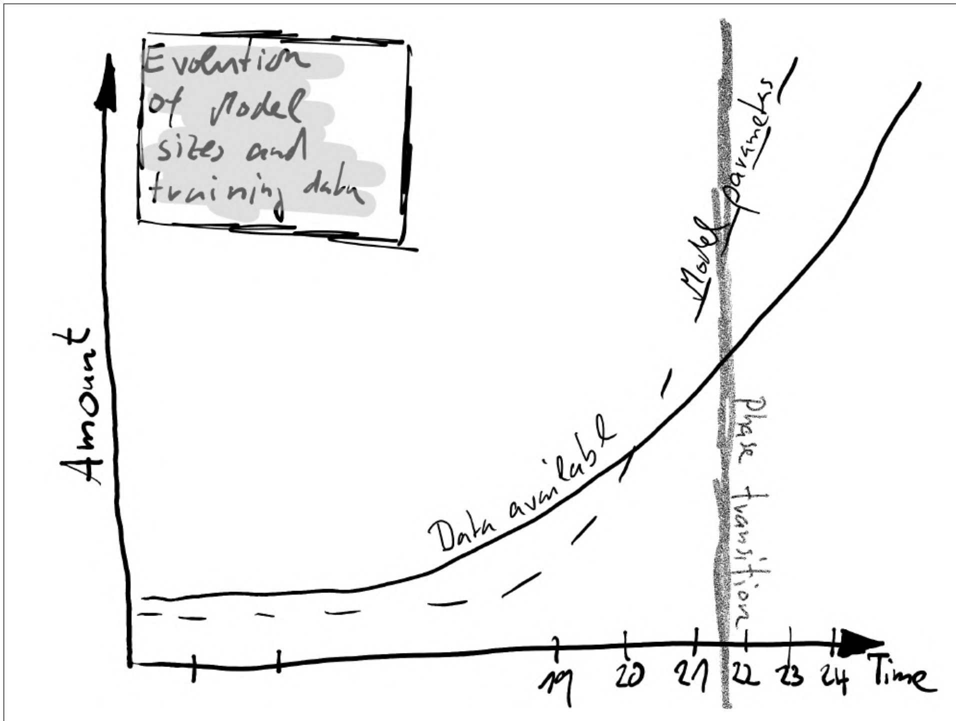


Figure 1-1. Explosion of parameters and training data lead to a phase transition.

Beyond just data, we owe this transformation to advancements in computational power, particularly the availability of specialized hardware like GPUs. This combinations of more data and faster compute created the perfect storm, enabling rapid advancements in generative AI models.

And with these advancements came new challenges, especially in managing the infrastructure required to handle such massive workloads. For example, as OpenAI detailed in their report on scaling Kubernetes to 7,500 nodes, Kubernetes emerged as a critical tool in managing the immense computational needs of models like GPT-3. Its ability to autoscale clusters, dynamically adjust infrastructure, and control costs made it an essential part of deploying these large models efficiently.

Most of us don't deal with clusters at the scale of OpenAI, but the underlying principles they developed are relevant for any Kubernetes environment, whether you're running LLMs on a small cluster or at "web scale."

As Kubernetes experts working on Red Hat's OpenShift, we were used to support traditional workloads - stateless applications, microservices, and databases - but running LLMs? That was a whole new ballgame. Our first experience with these computational monsters was both exciting and overwhelming. These models are like

"translucent" black boxes: we knew they were huge, needed GPUs, persistent volume space, and required health checks, but beyond that, the inner workings were opaque.

We still remember our first attempt vividly. It was a disaster. The models took ages to initialize, enabling GPU usage felt like falling down a rabbit hole, and CrashLoop-Back errors became our constant companions. The response times were embarrassingly slow. It became clear that we had to rethink how Kubernetes was handling these workloads.

After some trial and error, we managed to get things running. We fine-tuned our resource requests, optimized persistent volumes, and introduced smarter scheduling strategies to maximize GPU efficiency. Finally, the models sprang to life. It was a steep learning curve, but it highlighted the gap between Kubernetes' traditional strengths and the emerging needs of AI workloads.

Not everyone will face these exact challenges, but the lessons we learned are applicable to any Kubernetes environment. As the Kubernetes community continues to close the remaining gaps to make AI workloads, especially LLMs, first-class citizen, we invite you to join us on this journey. In this book, we'll explore the state of the art for running LLMs on Kubernetes and show you how to overcome the operational challenges that come with it.

In this introduction, we will first explore the challenges of running large AI workloads at scale. Next, we'll discuss why Kubernetes is such a powerful platform for addressing these challenges, despite a few gaps the community is actively working to close. We'll then take a brief detour to examine how the processes around operationalizing generative AI workloads have evolved, with a focus on how the DevOps paradigm has been specialized into MLOps, aimed specifically at managing machine learning workloads.

Finally, we'll provide an overview of the three key areas that will guide the rest of this book: Training, Inference, and AI-driven Applications.

Now, let's dive into the first critical topic: the challenges of running generative AI at scale.

Challenges running Generative AI at scale

As we have seen, running Generative AI models, particularly LLMs, involves navigating a set of complex challenges that extend beyond traditional application workloads. These challenges demand not just powerful hardware but also sophisticated management and orchestration of resources.

Generative AI and Large Language Models

In this book, we frequently use the terms "Generative AI" and "Large Language Models" (LLMs) interchangeably. Here's why:

Generative AI encompasses a wide range of techniques within the field of machine learning and artificial intelligence. This includes not only LLMs, but also models for generating images, videos, and sound.

While LLMs are just one subset of Generative AI, they have become the most prominent and widely recognized. To simplify our discussion, we'll often refer to both Generative AI and LLMs interchangeably.

Operationally, all Generative AI models share many similarities, though we will highlight any differences when necessary.

While most of the inner working of such models can be hidden for the operator, there are still requirements that makes AI workloads special:

Model Size and Resource Demands
: One of the most significant challenges in running LLMs at scale is their sheer size. LLMs consist of billions of parameters, making them resource-intensive in terms of both storage and memory. As these models grow in complexity and size, the need for efficient resource management becomes essential. The infrastructure must be capable of handling these models' demands without compromising performance or reliability. This is where the ability to dynamically allocate resources based on load and demand becomes crucial.

Start-Up Time and Latency
: Start-up time for these models can also be a bottleneck. Unlike traditional applications, LLMs require substantial warm-up periods, where their parameters are loaded into memory and optimized for inference. This latency can impact the overall responsiveness of AI-driven applications, making it essential to have systems that can manage start-up processes efficiently.

Hardware Requirements and Scalability
: Generative AI workloads are highly dependent on specialized hardware, particularly GPUs, which provide the necessary computational power for training and inference. Ensuring the right allocation of GPUs, managing their availability, and scaling services across multiple nodes is a challenge that requires advanced orchestration tools. Additionally, as models evolve, the infrastructure must support the integration of new hardware without disrupting ongoing operations.

Security and Data Privacy
: Security is another critical concern. LLMs are often trained on sensitive data, requiring stringent security measures to protect against unauthorized access and ensure compliance with data privacy regulations. The challenge is to implement security at multiple layers, from securing the data pipeline to ensuring that the models themselves are not vulnerable to attacks.

As you can see from this list, running Generative AI models at scale presents a complex set of challenges. These include managing enormous model sizes, addressing hardware requirements, and dealing with latency and security concerns. Each issue demands careful orchestration and robust infrastructure to maintain performance and stability. Without the right tools, these obstacles can become significant barriers to success.

Fortunately, Kubernetes provides a platform capable of handling these unique demands. In the next section, we'll explore how Kubernetes addresses these challenges and why it's an ideal fit for AI workloads.

Kubernetes for AI Workloads

We all know that Kubernetes is an open-source container orchestration platform developed by Google originally, now part of the Cloud Native Computing Foundation (CNCF). It was designed to automate the deployment, scaling, and management of containerized applications, which are packaged as OCI-compliant container images. Kubernetes abstracts the underlying infrastructure, allowing developers and operators to focus on deploying and managing applications without worrying about the complexities of the underlying hardware.

Initially, Kubernetes was optimized for distributed stateless workloads that can scale horizontally with ease. However Kubernetes quickly learned how to support stateful workloads, like databases and messaging systems. This evolution made Kubernetes a good platform for running a full stack of applications, from simple web services to complex, state-dependent systems.

In the context of AI, Kubernetes presents both opportunities and challenges. Traditionally, AI workloads, especially those involving large language models or other generative AI models, have unique requirements that differ significantly from typical business applications. These workloads often demand high-performance computing resources, and specialized hardware, such as GPUs. The challenge lies in extending Kubernetes to handle these demands effectively while maintaining its strengths in managing business applications.

In this book, we will explore how to leverage Kubernetes to operationalize generative AI models, addressing the specific challenges of running these workloads on a platform originally designed for more traditional applications. While we assume

some basic Kubernetes skills, we will delve into how Kubernetes' features can be used to support AI workloads and how additional Kubernetes addons and platforms like Kubeflow can help fill the gaps, particularly in areas like model training and inference.

Technology alone isn't enough though. Successfully running AI-driven applications on Kubernetes also requires a shift in how we think about application operations. This new mindset will be crucial as we integrate AI workloads into larger systems that also include traditional business applications. In the next section, we will discuss the evolution from DevOps to MLOps, highlighting how practices that revolutionized software development can be adapted to the AI domain.

DevOps and MLOps

DevOps emerged in the late 2000s as a response to the inefficiencies and bottlenecks that plagued traditional software development. In the past, development and operations teams often worked in silos, leading to misaligned goals, delayed releases, and frequent errors during deployment. DevOps seeks to bridge this gap by bringing these teams together. Beside the culture, DevOps is also as about best practices and tooling.

In summary, the most important aspects that are covered by DevOps are:

Collaboration
: DevOps emphasizes breaking down the barriers between development and operations teams. By sharing responsibility and open communication, DevOps ensures that both teams work together to create good software quickly and efficiently.

Automation
: Automation is at the heart of DevOps. It reduces manual errors and speeds up processes, ensuring more reliable outcomes. Automating tasks like testing and deployments frees up time for more creative work - and let's face it, writing scripts is more fun than following the same manual process over and over again.

Continuous Integration and Continuous Deployment (CI/CD)
: CI/CD is integral to the DevOps workflow. Continuous Integration involves the automatic testing of code changes as they are committed, while Continuous Deployment ensures that these changes are automatically released to production. This practice allows teams to deploy updates frequently and reliably.

Infrastructure as Code (IaC)
: DevOps promotes the management of infrastructure through code, allowing teams to define and provision computing resources using machine-readable con-

figuration files. This approach enables version control and ensures consistency in the deployment of infrastructure.

Observability
: In a DevOps environment, continuous monitoring of applications and infrastructure is crucial. Observability involves implementing logging, monitoring, and tracing systems to detect issues early and gather feedback, which is then used to improve the system.

The fluent interplay between developer oriented tasks and operational duties is visualized in Figure 1-2 as infinite loops that move between planning, coding, testing, releasing, deploying, and monitoring steps. This schema also emphasizes the integrated approach and shared responsibilities between previously clearly distinct roles.

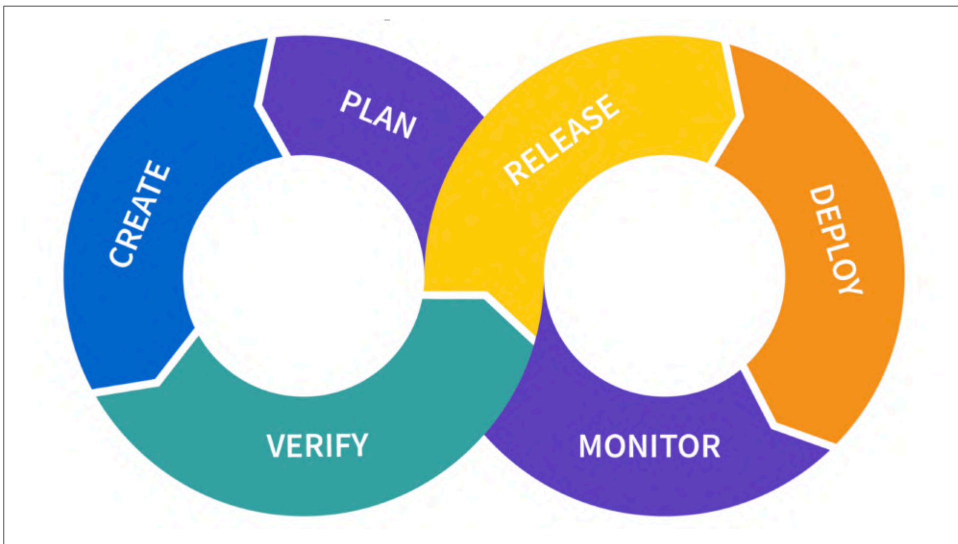


Figure 1-2. The inifinite DevOps loop

As software development has evolved, so too have the specialized practices that address the needs of new fields. One of the most significant developments has been the rise of MLOps (Machine Learning Ops), which extends the principles of DevOps to the lifecycle of machine learning models.

MLOps addresses the unique challenges of deploying, monitoring, and maintaining machine learning models in production environments. These additional challenges include:

Cross-functional Collaboration
: MLOps emphasizes the importance of collaboration between data scientists, machine learning engineers, and operations teams. Effective communication and

clear handover points are essential to ensure that models are properly integrated into production environments.

Comprehensive Versioning
: In addition to code, MLOps requires the versioning of data and models to maintain consistency and reproducibility across different environments.

Specialized CI/CD Pipelines
: MLOps adapts the CI/CD pipelines used in DevOps to accommodate the specific needs of machine learning models. This includes automated testing and validation of models before they are deployed to production.

Advanced Monitoring
: Monitoring in MLOps goes beyond traditional performance metrics. It involves tracking model-specific metrics such as accuracy, latency, and data drift to ensure that models continue to perform well over time.

Automated Model Management
: MLOps also involves automating the retraining and redeployment of models in response to changes in data patterns or performance degradation. This ensures that models remain accurate and relevant as they encounter new data.

This specialization adjusts the steps in our DevOps loop as shown in Figure 1-3: Coding involves crafting and architecting an ML model, testing focuses on verifying the usefulness, releasing entails packaging the model data into a suitable format (such as an OCI image in the context of Kubernetes), and deploying involves updating the runtime that serves the queries the model with the released model data.

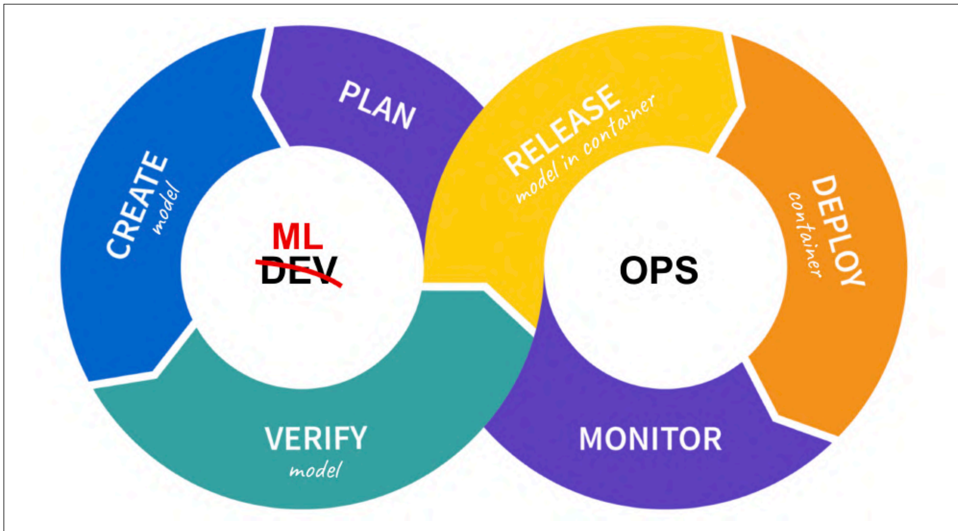


Figure 1-3. Adapting the DevOps loop for MLOps needs

The evolution from DevOps to MLOps highlights the growing complexity of managing machine learning workloads. It underscores the need for specialized tools and processes that address these unique challenges, ensuring that machine learning models in production are as reliable and efficient as traditional software systems.

Overview

As we discussed in "Challenges running Generative AI at scale" on page 13, running Generative AI on Kubernetes introduces a range of unique challenges that require innovative solutions. To effectively navigate these, we categorize the main tasks into three distinct areas: Training, Inference, and AI-driven Applications.

Training
: Finetuning from foundational models is one of the most resource-intensive tasks in the AI lifecycle. Kubernetes, with its scalability and resource management capabilities, is an ideal platform for spreading the load over many nodes for training large language models.

Inference
: Once a model is trained, the next challenge is deploying it at scale to serve predictions or generate content in response to queries. This involves setting up and managing an API for querying the model in a production environment, where performance and reliability are critical.

AI-driven Applications
: Kubernetes isn't just a platform for running models; it's a versatile application platform that can integrate LLMs into broader business applications. These applications often consist of multiple microservices, and integrating LLMs can enhance their capabilities—from automating tasks to providing advanced data insights.

Let's start with the inference phase, since even when it comes second in the AI model lifecycle, it's the most important and most common use case to operate a given LLM on Kubernetes.

Inference

The most common use case for running GenAI on Kubernetes is to offer querying the model as a servicer. This process is known as *Inference*. Inference involves using the trained model to generate predictions or outputs based on new inputs. To serve these models to a wide range of users, they must be deployed in a scalable and reliable manner. This is where Kubernetes shines, offering a robust platform for operationalizing inference at scale.

Kubernetes provides several key features that make it particularly well-suited for running inference workloads:

Declarative Resource Management
: Kubernetes allows you to define resource requirements declaratively, such as specifying the need for GPU acceleration or setting memory limits. Kubernetes then automatically schedules the model services onto appropriate nodes in the cluster.

Self-Healing Capabilities
: Kubernetes continuously monitors the health of your model services. If a service fails or becomes unhealthy, Kubernetes can automatically restart it, ensuring high availability and reliability.

Containerization
: Containers are an ideal way to package and version models. They provide a consistent environment for model execution, regardless of the underlying infrastructure, making deployment and scaling more manageable.

Fine-Grained Access Control
: With Kubernetes Role-Based Access Control (RBAC), you can implement granular policies that define who can manage, access, or modify your model services, ensuring security and compliance.

Extensibility with Add-ons
: Kubernetes supports extensions such as KServe, which offers a dedicated abstraction layer for serving machine learning models. KServe simplifies the deployment and management of model services, providing features like autoscaling, canary rollouts, and built-in monitoring.

??? dives deeper into these topics, exploring how to leverage Kubernetes to serve models in a production environment, ensuring they are scalable, reliable, and secure.

Training

While deploying models for inference is crucial, creating those models from scratch is an entirely different challenge - one that is both resource and cost-intensive. Only the largest companies, with the necessary financial backing, can afford to train large language models from the ground up. The sheer volume of data required, combined with the computational power needed, makes this challenge nearly impossible for most organizations.

As a result, most teams turn to foundational models - pretrained models provided by large companies. These models serve as a starting point for further customization and are made available under various licenses, which often include restrictions on commercial use.

To give you an idea of the options available, Table 1-1 shows some well-known open-source LLMs along with their sizes and parameter counts:

Table 1-1. Sample models and their sizes

Name	Vendor	Parameters	Size
Llama 405B	Meta	405 billion	~750 GB
OPT-175B	Meta	175 billion	~350 GB
Vicuna	LMSYS	33 billion	~66 GB
Orca	Microsoft	13 billion	~26 GB
Granite 13B	IBM	13 billion	~26 GB
Falcon 2	TII	11 billion	~22 GB
LLaMA 3	Meta	8 billion	~16 GB
Mistral 7B	Mistral	7 billion	~14 GB

While foundational models are powerful, they typically lack specialized knowledge in domains that aren't publicly accessible. This is where fine-tuning comes into play. Fine-tuning is the process of adapting a foundational model to your specific needs by training it further on a targeted dataset. Techniques like Low-Rank Adapters (LoRA) enable this process to be more efficient, reducing the computational resources required.

Kubernetes excels in facilitating fine-tuning within a cluster environment. By leveraging Kubernetes' scalable infrastructure, you can run the fine-tuning process efficiently, taking advantage of distributed computing resources to handle the intensive workload.

All the details of these processes will be explored in ???.

CHAPTER 2

Deploying Models

> ### A Note for Early Release Readers
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

Have you experimented with online models like OpenAI's ChatGPT, and explored prompt engineering to get useful content from the model, but now you need to run the model within your own cluster because your real data can't leave it? If so, then this chapter is for you!

There are many different models on the market, many of them are open source and available online with a permissive license. Hugging Face is the most famous community where you can find not only models but also dataset and libraries.

Regardless of where you obtained the model, whether it's open-source or not, there are aspects of deploying the model on Kubernetes that aren't specific to the model itself. However, some aspects require careful analysis of the model to determine the best approach.

This chapter describes different approaches and patterns for managing the lifecycle of your model at runtime, with a focus on some of the most used runtimes for *Large Language Models (LLMs)*.

Transformer architecture and Attention mechanism

Generative AI is a vast field, and the maturity levels of different model classes vary significantly, with text generation models being the most widely used and optimized.

Text generation models are based on *Transformer architecture* or a derivative (like *Mixture of expert* approach) and they can cover multiple use cases (task) that involve the processing of text: chatbot, code generation, translation, summarization, etc.

Transformer architecture is a deep learning architecture created and introduced by Google in 2017 to be more efficient in long-range dependencies tracking via Attention mechanisms. The main advantage of this architecture, compared to others, like recurrent neural networks (RNN), is that it doesn't have recurring units (i.e., using the output of one neuron as the input to another). This makes it highly parallelizable during training.

Long-range dependency is a core concept in natural language processing: the meaning of a sentence is influenced by the context.

The attention mechanism is used to to mimic human attention by assigning different weights (or importance) to various components of a sentence (or vector). In particular a multi-head attention mechanism is used to run an attention mechanism in parallel several times to produce different outputs that are then finally concatenated and linearly transformed.

For more information on Transformer architecture and attention see "How do Transformers work?"

"It works on my machine"

In a nutshell, deploying a model requires both the model itself and a runtime capable of loading and executing it. As mentioned, Transformer-based models are the most common Large Language Models. Therefore, you can use the Transformer library from Hugging Face to load the model and invoke it. This doesn't mean that every laptop can handle a similar workload and neither that model of every size can be loaded: it is possible to execute some models using CPU with very limited performance (tens of second to produce a full sentence) thus a GPU is essentially required. Moreover memory requirements are directly related to the size of the model: a model with 7 billions of parameters (aka 7B) is considered a *Small Language Model (SLM)* and requires a GPU with about 15GB of memory to be loaded while a 70B model requires about 140GB of memory.

See Example 2-1 for a code snippet with illustration purposes.

Example 2-1. Load and execute locally Llama3 8B

```
import transformers
import torch

model_id = "meta-llama/Meta-Llama-3-8B"  ❶

pipeline = transformers.pipeline(        ❷
    "text-generation", model=model_id,
    device_map="auto"
)
pipeline("Hey how are you doing today?") ❸
```

❶ The model identifier in Hugging Face format.

❷ Load and initialize the model.

❸ Invoke the model with a prompt.

What's next? We'll want to make the prompt the user input and expose the function through an endpoint.

Let's go back to Example 2-1 to see how we can make it more flexible accepting the prompt with an endpoint to make it more similar to a real world scenario. The easiest improvement is to avoid the download of model on the fly every time the runtime (or a replica) is started. The pattern to download and initialize the model is quite common during the development/experiment phase but it is possible, and usually suggested, to make the model available to the cluster without the need to access internet. There are different file formats, storage options and loading techniques, see Chapter 3, "Model Data" for more information.

The next step is to expose the model with an endpoint so that the prompt is dynamic and that multiple users can invoke it. One simple way to do that is to leverage Python ecosystem and in particular FastAPI and Pydantic. See Example 2-2.

Example 2-2. FastAPI generate endpoint

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class InputText(BaseModel)
    text: str
```

```python
class OutputText(BaseModel)
    text: str

pipeline = get_pipeline() # see previous example

@app.post("/generate", response_model=OutputText)
async def generate_func(prompt: InputText):
    output = pipeline(prompt.text)
    return {"text": output[0]["generated_text"]}
```

Can we just create a container image and deploy it on Kubernetes?

As you can imagine it is not that simple, especially if you are preparing your Kubernetes cluster for a production workload where scalability/throughput, reproducibility, and monitoring are critical.

At the same time the example is not really model specific so it looks like we are already creating something generic and that might be generalized even more. Essentially, we are recreating a model server!

Model Server

A Model Server (or serving runtime) is a component that includes one or more runtimes. It can be distributed to use multiple GPUs at the same time, execute various types of models exposed via an API (REST or gRPC) and is optimized to maximize throughput (Figure 2-1)

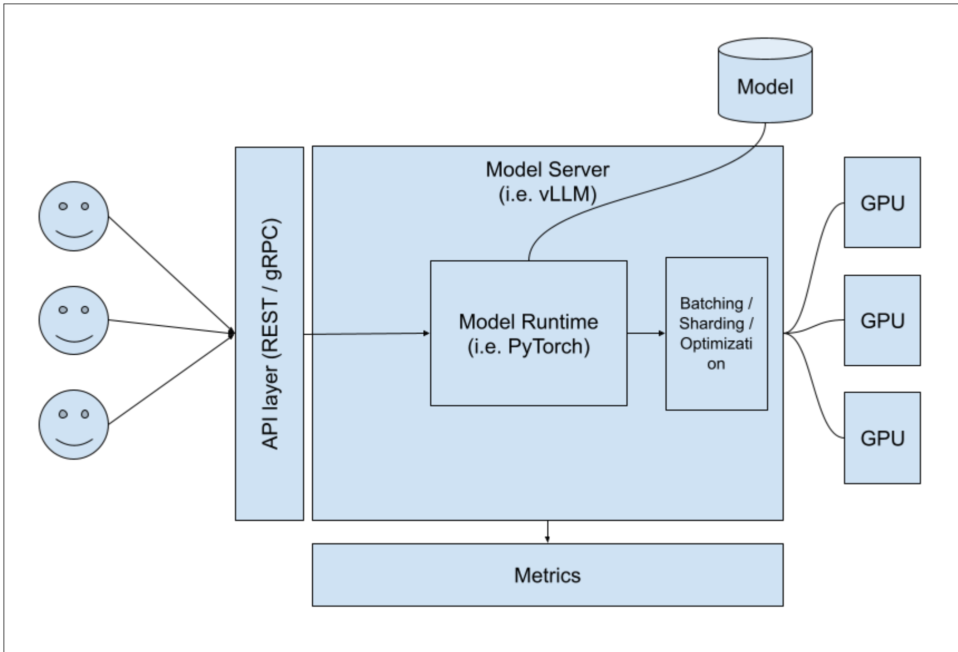


Figure 2-1. Model Server architecture

This concept is not new or specific to Generative AI, there are multiple existing model servers that uses the common frameworks to serve any type of Predictive AI model and some of them are also evolving to support Generative AI. Even if the concept is the same the exposed API is very different. In Predictive AI the endpoint is usually a generic `/predict` or `/infer` because the model is considered as a black box function while in Generative AI it is a more task oriented API because similar models can perform different types of actions and work with different types of modalities (multimodal): text generation, summarization, classification, text to image, etc.

Model Servers expose the AI model via an API that clients have to use. This API can be specific for a particular model server implementation breaking the abstraction that Model Server aims to provide because client applications should not be tied to a specific implementation.

This problem is not new nor specific to Generative AI, for Predictive AI the KServe open-inference-protocol (OIP) has been defined as specification to standardize "infer" endpoints and it has been adopted by most of model servers and is now expanding to include Generative AI.

The API to invoke Generative AI models are still overall experimental and very different based on the type of model and the task it performs. OpenAI with chatGPT API for chat completion is a standard de facto for text generation models.

Multimodal models

Many LLMs typically work with just one modality: input and output are text. Multimodal models are able to process a larger set of modality like images, video, audio, mathematical equations and so on. In particular the main goal is to mix similar modality to perform tasks like text to image where the input is a textual query and the output is a generated image. It's possible to do the opposite or to mix multiple modalities in the same query by providing an image and a query to return a new image or text.

From a model architecture perspective image/audio generation models are very different compared to text generation models: they are *diffusion models* and not *Transformer* based. This category of models is part of the Generative AI space but it is not a Large Language Model, they are currently adopted mainly for specific departments like for image generator/editing for marketing and there is less standardization around. They usually directly integrated in other specialized product like image editor solutions.

We assume in the book the usage of Large Language Models *Transformer* based that are applicable to a larger set of use cases. This implies that the model output is text but it doesn't prevent the input to include images / audio together with text making them multimodal models.

From a platform/Kubernetes perspective every model server is usually similar in terms of deployment topology but you should be aware of the type of model and task because the scaling, hardware optimization and metrics to observe are model server specific. We'll delve more into this content in Chapter 4, "Model Observability".

Now that we know what a model server is, let's go more in details with few examples of LLM model servers, including an example and highlighting there main use case.

vLLM

vLLM is a LF AI & Data project for LLM inference and serving. The project is very active, with thousands of forks, hundreds contributors, the support of more than 50 model architectures, end-to-end optimization techniques and the support of multiple hardware vendors. It is a library that can be directly used in Python (Example 2-3) but the project includes a CLI and an OpenAI-compatible server.

Example 2-3. Load a model in vLLM and execute inference

```
from vllm import LLM

llm = LLM(model="meta-llama/Meta-Llama-3-8B")  ❶
results = llm.generate("LLMs are great for")   ❷

print(results[0].outputs[0].text)              ❸
```

❶ Load model

❷ Invoke the model

❸ Extract result

Our goal is to serve the model on Kubernetes, so vLLM should be run in a container, making a server the best option. Starting the server requires minimal configuration. However, a key difference to note is that in a production Kubernetes environment, you will likely use a local copy of the model rather than fetching it on-the-fly from Hugging Face. You'll need to specify the location of the local model to the server. See Example 2-4.

Example 2-4. Use vLLM server

```
# start the server
vllm serve \                                       ❶
 --port=8080 \
 --model=/mnt/models \                             ❷
 --served-model-name=meta-llama/Meta-Llama-3-8B    ❸

# invoke the model
curl http://localhost:8080/v1/completions \
 -H "Content-Type: application/json" \
 -d '{
  "model": "meta-llama/Meta-Llama-3-8B",
  "prompt": "LLMs are great for",
```

```
  "max_tokens": 10,                              ❹
  "temperature": 0                               ❺
 }'
```

❶ It is equivalent to `python -m vllm.entrypoints.openai.api_server`

❷ Location (or name) of the model (local to the container)

❸ Name of the model

❹ Number of tokens the model should produce

❺ Temperature controls the randomness of the sampling, 0 makes the generation deterministic

From a platform/Kubernetes perspective, many parameters are used to configure how the runtime loads and executes the model, but this is relatively transparent from a deployment standpoint. Techniques such as PagedAttention, FlashAttention, and speculative decoding are focused on efficient attention management and faster execution. While these techniques don't impact deployment directly, they do affect scalability and resource optimization.

LLM inference optimization

The optimization of LLM execution is a very active field with new techniques every weeks, this is the area where academia and engine implementation are strictly coupled.

It is almost impossible to keep up with the speed of evolution and at the same time it takes time to properly measure/assess if a new optimization provides the expected benefit or not.

In this scenario we already mentioned some key optimizations like PagedAttention and Flash Attention specific to make self-attention faster given the quadratic time and memory complexity of this phase optimizing memory management.

Another investment area is to reduce the size of the model minimizing performance loss using multiple quantization techniques to reduce the floating point size of the weights of the model.

The cost and the complexity to produce a token is not the same for every token, in natural language there are tokens that are very common and easy to predict so why don't exploit this aspect to reduce the execution cost? Speculative decoding is an optimization techniques based on this principle.

As you can see there are many different way to optimize the execution of a LLM, this book doesn't aim to explain all of them but fortunately, from a MLOps engineer

perspective, you don't need to be an expert in LLM optimization internals, it is critical to use a Model Server that is actively developed with a large community so that every new optimization is included. The configuration of vLLM for example, is usually limited to changing the startup parameters of the runtime and the project is getting better and better to automatically detect, based on the model to execute, which configuration to apply so most likely the default values should work.

Some of the configuration like quantization has effect on the quality of the model and the tuning to find the right trade off are part of the model development/tuning so that at inference time you should already get the configuration as part of the deployment.

On the other hand as an MLOps engineer you should be aware of the parameters that have larger implication on parallelization and scaling: multi node/distributed serving has an impact on overall topology, it usually requires additional components to manage the coordination and makes the deployment stateful. We will discuss running the model in more detail in Chapter 6, "Running In Production".

Hugging Face Text Generation Inference

Hugging Face Text Generation Inference (TGI) is another Open Source model server implementation created by the Hugging Face company to serve text generation models and it is used to power their product offering. Hugging Face has been mentioned multiple times already because it is the most active community where you can share Generative AI models (base or fine tuned models) but also datasets and libraries. Many of the most used libraries used for Generative AI, like `transformer`, `peft` or `diffusers`, are incubated in this community.

Similar to vLLM it has a launcher that can be used to start the server and load the model. See Example 2-5.

Example 2-5. Use text generation inference server

```
# start the server
text-generation-launcher \                    ❶
 --port 8080 \
 --model-id /mnt/models                       ❷

# invoke the model using TGI API
 curl localhost:8080/generate_stream \        ❸
    -H 'Content-Type: application/json' \
    -X POST \
    -d '{"inputs":"LLMs are great for",
     "parameters":{"max_new_tokens":10}
     }'

# invoke the model using OpenAI-compatible API
```

```
curl localhost:3000/v1/chat/completions \   ❹
    -H 'Content-Type: application/json' \
    -X POST \
    -d '{
  "model": "tgi",
  "messages": [
    {
      "role": "system",                      ❺
      "content": "You are a helpful assistant."
    },
    {
      "role": "user",
      "content": "LLMs are great for"
    }
  ],
  "max_tokens": 10
}'
```

❶ Launcher command

❷ Location (or name) of the model (local to the container)

❸ TGI original API to invoke the model

❹ TGI now supports also OpenAI-compatible API

❺ One of the most common categories of fine-tuned models is "instruct" models, which are designed to follow human instructions. In this scenario, the system prompt defines the role of the model.

The same comments about the parameters and their implication on Kubernetes made to vLLM applies here.

Other model servers

llama.cpp, as the name might suggest, is a `C++` implementation that runs Llama models.

It was originally created as a full re-implementation of the Transformer architecture in C++ specifically for Llama models. Over time, it has evolved to support a variety of other models. The focus has been on efficiency, making it the recommended option for running similar models locally on a laptop. Although it still requires a powerful machine, it is widely used by projects such as Ollama, Ramalama, LM Studio, and InstructLab. While it is not designed for production use cases with high concurrency, an active community continues to reimplement many optimizations and techniques in C++, making llama.cpp increasingly powerful. One of the results of

the development of llama.cpp has been the creation of GGUF file format that now has been adopted by other libraries too.

In addition to the core library, there is a python server that exposes OpenAI compatible API similar to the other model servers, see Example 2-6.

Example 2-6. Start llama.cpp python server

```
python -m llama_cpp.server \        ❶
 --model /mnt/models                ❷
```

❶ Start llama.cpp server

❷ Location of the model (local to the container)

NVIDIA is the leader provider of GPU for AI and it also provides the necessary software to train and serve models. NVIDIA NIM is a solution designed for Kubernetes provided by NVIDIA to simplify the deployment and the optimization of a LLM on their hardware. It includes a model server (NVIDIA Triton Inference Server) but it takes a different approach with a curated container image per model family. This means that models are directly tested and published by NVIDIA so you need to check the supported model list (like Llama and Mistral) in the documentation. As you can see, this approach is less flexible. However, NVIDIA NIM stands out as a model server due to its opinionated design regarding model and hardware usage, offering some notable features: local caching of the model and hardware optimization. Local caching is supported by a `PersistedVolume``, aiming to simplify and speed up one of the major pain points of model serving for LLMs: loading time. The model is downloaded only once, and subsequent replica creations or restarts do not trigger another download. Hardware optimization is another key feature, allowing NVIDIA NIM to detect available accelerators, select the most suitable model for the configuration, and adjust the model server settings accordingly. See Figure 2-2 for more details on NVIDIA NIM Architecture

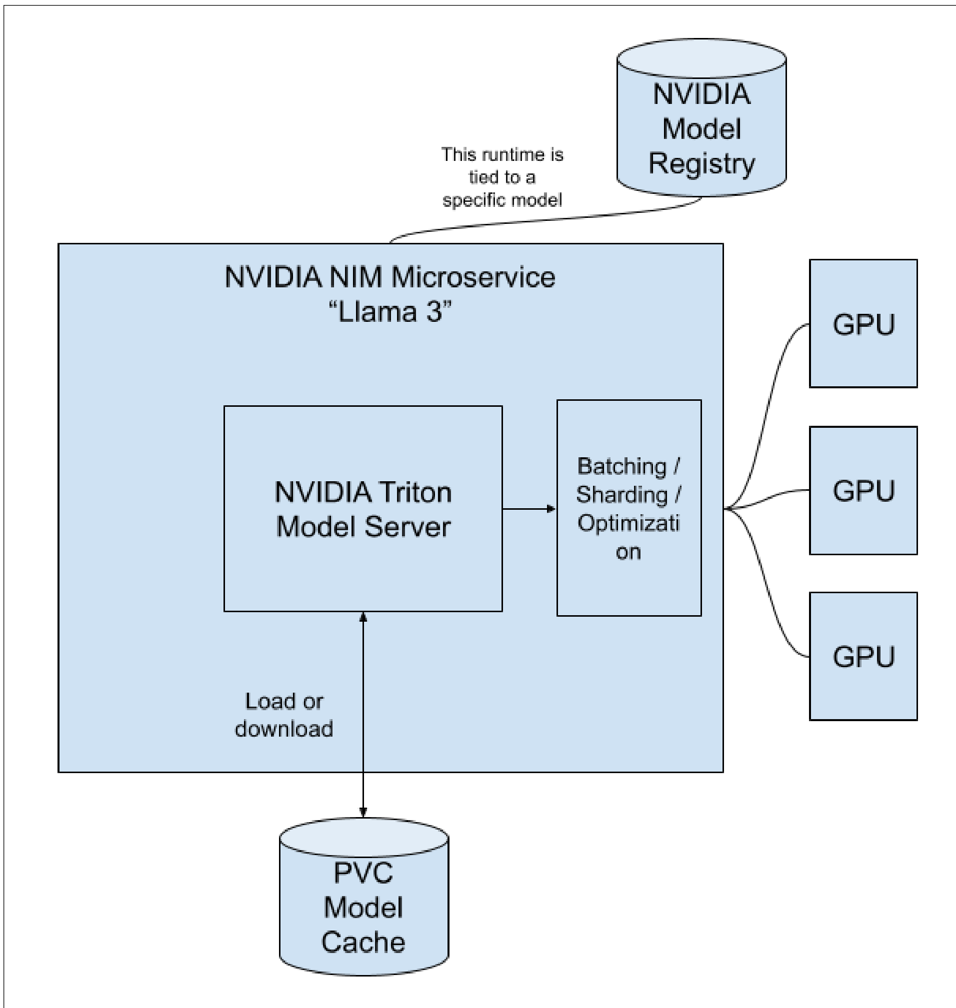


Figure 2-2. NVIDIA NIM Architecture

Model Server Controller

Now that we understand what a model server is, how to use it to serve a model, and some of the model servers specialized for LLMs, we are ready to introduce the final step of integration with Kubernetes: deployment.

You need to have an image that includes the model server and then create a deployment that integrates all components: the model server, model, accelerator, and observability. Figure 2-3 extends the previous Model Server architecture diagram to include the main controller components: one or more Kubernetes CustomResource-Definition and a Kubernetes Controller.

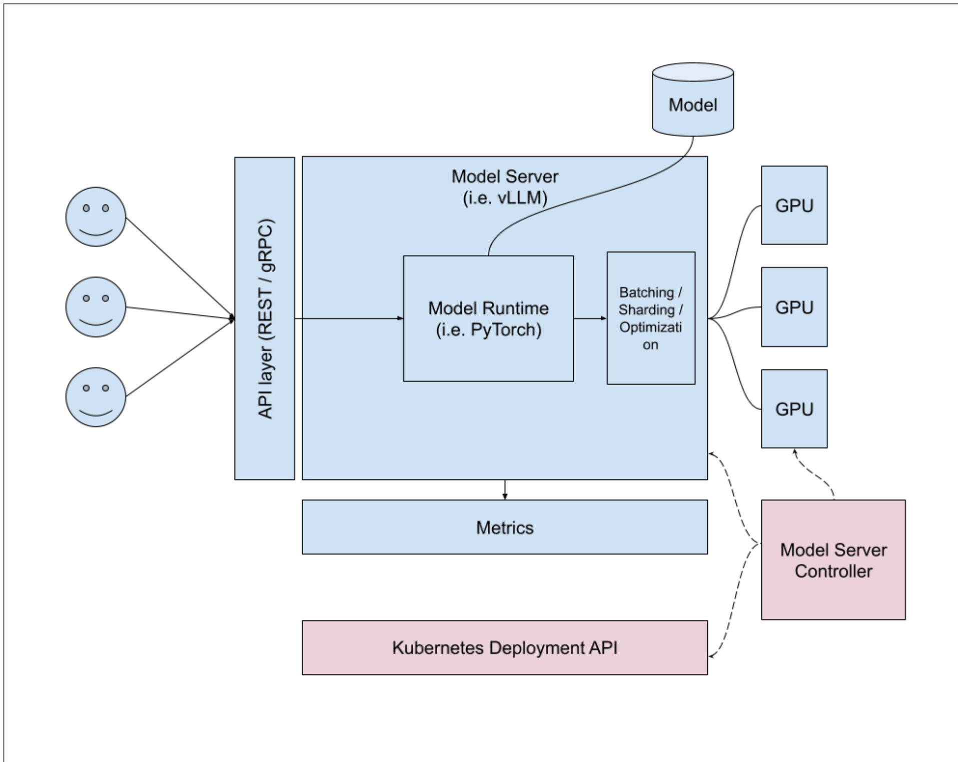


Figure 2-3. Model Server Controller architecture

Each model server usually provides the images so that you don't need to build it but at the same time picking the right image is not straightforward: each accelerator has different driver/framework (i.e. NVIDIA with CUDA, AMD with ROCm, etc) so it is necessary to pay attention to this aspect. This concern is similar to multi-architecture containers, where you can easily select the architecture (e.g., `arm64`` or `i386``) and get the appropriate container version. However, for accelerators, the process is still quite manual, so it's important to pay attention to this aspect.

DIY - Do It Yourself

The DIY option is always available and sometimes necessary if you need to customize every aspect of the deployment in a controller environment.

Let's assume you want to use vLLM and you already built/got the image to use. If we look at Example 2-7 you can easily spot most of the configuration that you need to consider in your deployment: port to expose, path where the model is and GPU configuration and parameters to execute the model that are essentially model specific.

Example 2-7. Start vLLM server with GPU

```
# specify which of the available GPUs to use
CUDA_VISIBLE_DEVICES=0,1
vllm serve \
 --port=8080 \
 --model=/mnt/models \
 --served-model-name=meta-llama/Meta-Llama-3-8B
```

Now that we know how to create a deployment, the name of the model we want to use, and the GPU requirements, we are ready to proceed. See Example 2-8 for the full `Deployment` spec (with `PersistenceVolumeClaim`) to apply to your cluster.

Example 2-8. Deploy vLLM server with GPU

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: vllm
spec:
  replicas: 1
  template:
    spec:
      containers:
        - resources:
            limits:
              cpu: '4'
              memory: 12Gi
              nvidia.com/gpu: '1'                         ❶
            requests:
              cpu: '2'
          name: vllm
          env:
            - name: HUGGING_FACE_HUB_TOKEN                ❷
              value: ''
          args: [                                         ❸
            "--port",
            "8080",
            "--model",
            "meta-llama/Meta-Llama-3-8B",
            "--download-dir",
            "/models-cache" ]                             ❹
          ports:
            - name: http
              containerPort: 8080                         ❺
              protocol: TCP
          volumeMounts:
            - name: models-cache
              mountPath: /models-cache
          image: vllm/vllm-openai:latest
      volumes:
```

```yaml
          - name: models-cache                        ❻
            persistentVolumeClaim:
              claimName: vllm-models-cache
        tolerations:
          - key: nvidia.com/gpu                       ❼
            operator: Exists
            effect: NoSchedule
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: vllm-models-cache
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 100Gi
```

❶ In addition to the traditional CPU/memory you can specify the number of GPU the model needs

❷ One option with vLLM is to download the model on-the-fly from Hugging Face, which requires injecting the token as an environment variable

❸ The entrypoint of the vLLM image is already starting the server so it is only necessary to specify the additional parameters

❹ In the scenario of download on the fly, it is suggested to specify a persisted model cache where the model is stored.

❺ The port to expose (useful then to expose it via `Service` and `Ingress`)

❻ The volume persisted volume to use as cache

❼ Tolerations and Taints are used to mark the nodes where the GPUs are available and make sure Kubernetes is going to schedule the deployment accordingly

This example is definitely not intended to be comprehensive: it doesn't cover the configuration of GPU in Kubernetes (more of this will be covered in Chapter 6, "Running In Production"), restart policy, scaling and even probes are missing. It is also limited to scenarios where the model can be deployed to a single node and not the distributed serving scenario. At the same time it is quite self contained and straightforward to start with. When the size of your organization and the configura-

tion of the different concerns make this solution hard to manage you can consider the introduction of additional components like KServe or KubeRay.

KServe

KServe project is Model Inference Platform on Kubernetes designed to manage the lifecycle and the wiring of model servers and models leveraging Kubernetes components to provide scalability, routing, canary rollout, density packing and in general the possibility to compose/extend a model inference endpoint.

The project has been created as part of Kubeflow community as KfServing years ago and then became independent (but still part of Kubeflow ecosystem). The initial target has been Predictive AI and only more recently evolved to include Generative AI.

KServe is built as a Kubernetes native component extending Kubernetes API providing multiple `CustomResourceDefinitions` to map the different concepts in a declarative way. We are not going to cover all the API and concepts that KServe provides because most of them are still mainly applicable to Predictive AI.

From a technology stack perspective there are three different deployment mode: Serverless, RawDeployment and ModelMesh.

Serverless
: Serverless is the most comprehensive stack, it uses Knative and Istio to manage autoscaling, rolling updates, traffic management and composition (also via Knative Eventing). By using this mode, every model becomes a `Knative Service`.

RawDeployment
: RawDeployment is the opposite of Serverless, with no additional dependencies beyond what Kubernetes already provides. Using this mode, for every model KServe creates a new `Deployment`.

ModelMesh
: The ModelMesh solution is specialized for high-density deployments where you need to deploy many models—potentially thousands—in the same cluster, and the footprint of using separate `Deployments`` is too large. In this mode the model server is dynamically loading and unloading models based on the requests.

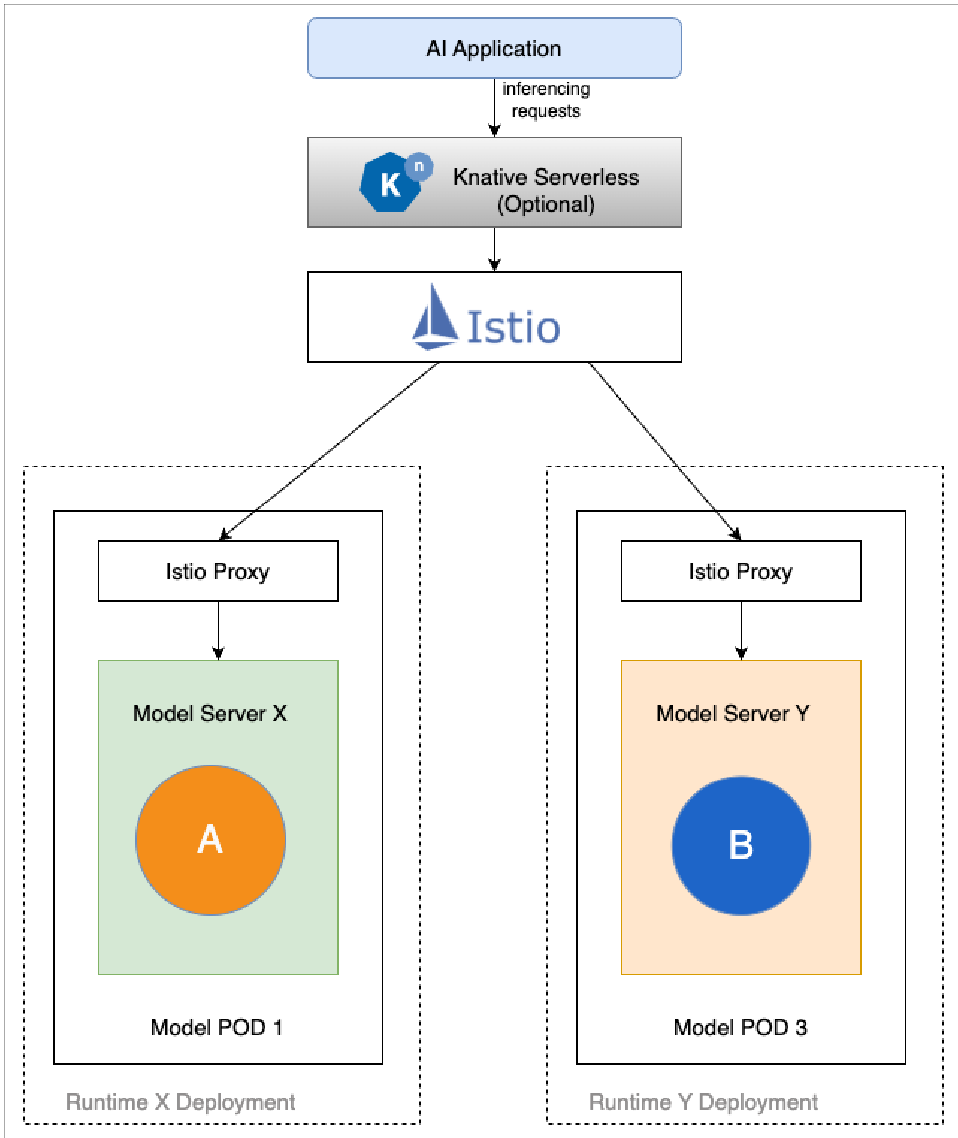


Figure 2-4. KServe Serverless and RawDeployment architecture

The last deployment mode is not really applicable to Generative AI: the size and the complexity of similar models doesn't really give you the option to deploy multiples of them in the same node. On the other hand, the other two deployment modes are generally applicable to Generative AI. However, the hardware requirements for these models are still largely managed statically, making it challenging to leverage the

dynamic autoscaling advantages of Serverless mode for LLMs. For the remainder of this section, we will assume `RawDeployment` as the deployment mode.

The two main APIs that KServe provides to deploy a model are `ServingRuntimeime` and `InferenceService`.

ServingRuntime
: A `ServingRuntime` is equivalent to a template/podSpec where a model server is declared in the `Namespace`. It specifies the image of the model server to use, along with some parameters and the type of model it can serve. This concept aims split and simplify runtime configuration and model configuration so that the owner of the project can have better control of model server versions, default configuration and overall centralize the lifecycle of the runtime. It is also possible to use a `ClusterServingRuntime` to configure a runtime that is available for the whole cluster. See Example 2-9.

Example 2-9. ServingRuntime example

```
apiVersion: serving.kserve.io/v1alpha1
kind: ServingRuntime
metadata:
  name: vllm                                    ❶
spec:
  containers:                                   ❷
    - args:
        - --model
        - /mnt/models/
        - --port
        - "8080"
      name: kserve-container
      image: vllm/vllm-openai:latest            ❸
      ports:
        - containerPort: 8080
          name: http1
          protocol: TCP
  multiModel: false
  supportedModelFormats:
    - autoSelect: true
      name: pytorch                             ❹
```

❶ KServe includes a vLLM ServingRuntime pre-configured named “HuggingFace Runtime” designed to serve all HuggingFace models. It can be used as is or you can define your own ServingRuntime using a similar specification

❷ This is the podSpec where it is possible to configure all the parameters necessary to run the model server

❸ This is the image that will be used. Note: applying this resource will not deploy the model server immediately, but it will make it available within the Namespace for use.

❹ vLLM, like most of the model server, uses PyTorch as actual runtime for the model so this configuration declares that this runtime is able to serve PyTorch models

InferenceService
: An `InferenceService` represents the model that the user wants to serve. This object can specify a `ServingRuntime` to use or the selection can be automatic based on the model format. The creation of this resource is going to trigger the deployment of the model server and the wiring of the model. In the same spec it is possible to override the default parameters specified in the `ServingRuntime` and add more configuration that might be specific for the model. See Example 2-10.

Example 2-10. InferenceService example

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: Meta-Llama-3-8B
  annotations:
    serving.kserve.io/deploymentMode: RawDeployment        ❶
spec:
  predictor:
    model:
      modelFormat:
        name: pytorch                                       ❷
      runtime: vllm                                         ❸
      storageUri: pvc://llama/model                         ❹
    containers:
      resources:                                            ❺
        limits:
          cpu: "4"
          memory: 50Gi
          nvidia.com/gpu: "1"
        requests:
          cpu: "1"
          memory: 50Gi
          nvidia.com/gpu: "1"
```

❶ This annotation is to select the deployment mode

❷ Declaring the type of model allows KServe to automatically find a `ServingRuntime` that can handle it

❸ It is also possible to specify the name of the runtime to refer explicitly

❹ This field specifies where to get the model, in this case from a PVC local to the cluster

❺ For each model it is possible to override the resources to match the requirements of the model

Other concepts/API

KServe API is very flexible and includes many other concepts that are not strictly necessary to deploy a LLM but that enables more advanced and composable use cases. It is possible to configure an inference logger to forward every input/output of the model to a logger service for auditing or training purposes, do some pre/post processing using a "transformer" (this is not related to Transformer architecture, it is just a name clash) or even compose different models using an `InferenceGraph`. See KServe Control Plane API for a more comprehensive documentation.

One of the main benefits of the split between `ServingRuntime` and `InferenceService` is a more defined ownership in terms of management because the runtime lifecycle and model lifecycle are very different. KServe also provides additional benefits like the support of multiple storage options: KServe controller inject an `initContainer` called storage initializer that reads the location of the model, performs the download (if necessary) and copies the model to a folder of the model server. It is also possible to replace the storage initializer container using the `ClusterStorageContainer` API with a custom one to support custom protocols for centralizing catalog of available models. We will cover more in details how to package, register and load a model in Chapter 3, "Model Data"

Ray Serve and KubeRay

The Ray project, compared to KServe, is a newer project with a broader scope. It is an open-source framework designed to build and scale ML applications easily. It is very Pythonic, making it user-friendly for those with Python experience, and allows you to configure all activities directly within your Python codebase.

Ray is not specific for model serving but instead it defines a set of core concepts quite generic: Task, Actor, Object, Placement Group and Environment Dependency. These core concepts in addition to the Ray Cluster define the execution model that is used to build and scale all the other features.

If you need a more comprehensive foundation on Ray, we suggest *Learning Ray* by Max Pumperla, Edward Oakes, Richard Liaw (O'Reilly Media)

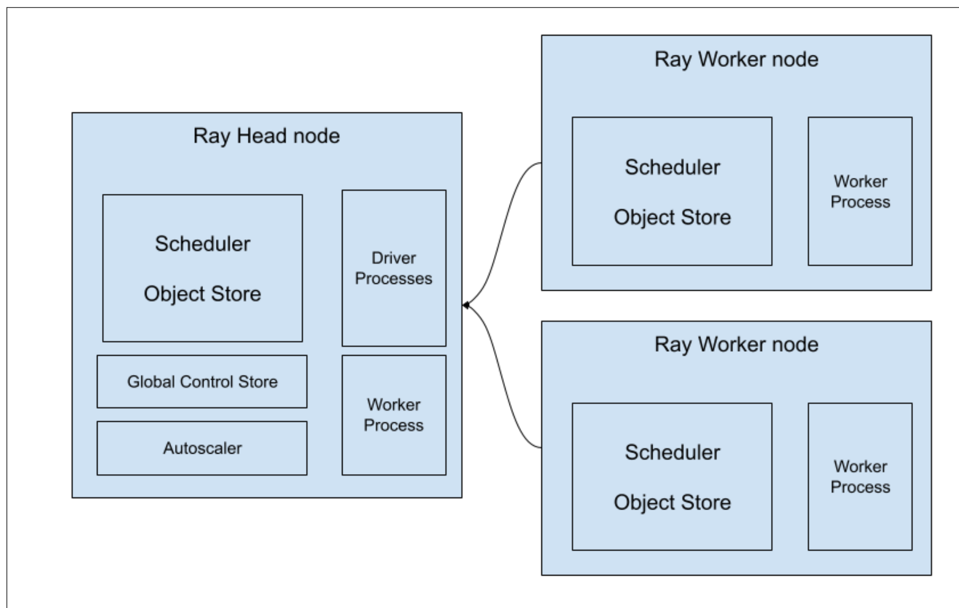


Figure 2-5. Ray Cluster topology

As you can see from the diagram, a Ray Cluster has not been designed with Kubernetes in mind and indeed it has a standalone infrastructure to manage the scheduling and orchestration of the jobs that you can usually do with Kubernetes API and the different worker nodes. There is the concept of Head node that acts as entrypoint for the jobs that are then dispatched to one or more Worker nodes where the execution will happen.

The set of features that Ray offers covers most of the ML use cases: Ray Train, Ray Tune and Ray Serve are just a subset of them. Ray Serve is the component that we need to use to serve a model, the deployment is defined in Python and same for each endpoint to expose or the initialization of the model. Example 2-11 is a very simplified scenario where a Transformer model is configured and deployed, in a way very similar to the first section of this chapter. Ray Serve given that is configured directly in the code is very flexible, you can easily find examples where it is integrated with FastAPI to expose the endpoint or using a library like vLLM to deploy a full model server.

Example 2-11. Ray Serve with a Transformer based model

```
from starlette.requests import Request
from typing import Dict

from transformers import pipeline

from ray import serve

@serve.deployment                                           ❶
class TransformerModelDeployment:
    def __init__(self):
        self._model = pipeline(                             ❷
            "my-transformer-model")

    def __call__(self, request: Request) -> Dict:
        return self._model(
            request.query_params["text"])[0]


serve.run(                                                  ❸
    TransformerModelDeployment.bind(),
    route_prefix="/my-model/")
```

❶ Decorator function where it is possible to configure most of the deployment aspects like autoscaling

❷ The init method should be used to load a model, in this case it is a Transformer-based pipeline

❸ This method deploys the model with a given prefix

Ray has an API that is very friendly to a Data Scientist or in general a Python developer, but when it comes to deploy a Ray Cluster on Kubernetes you still need some help to wire all the components together with Kubernetes concepts like Deployment and Ingress.

The KubeRay project has been created to make the transition from local Ray execution to Kubernetes streamlined. This is necessary because Ray clusters and Ray applications are not natively designed to use Kubernetes, in particular a Ray cluster has a head node and worker nodes that needs to be deployed with multiple Deployments properly configured to interact each other.

KubeRay provides multiple Ray API as Kubernetes CustomResourceDefinition, but in particular the `RayService` object is a single concept that represents a multi node Ray Cluster and a Ray Serve application that uses that cluster. Example 2-12 is not a full example of the spec but it highlights the main elements of the spec.

Example 2-12. RayService CR snippet

```
apiVersion: ray.io/v1alpha1
kind: RayService
metadata:
  name: my-transformer-model
spec:
  serveConfigV2: |                                                    ❶
    applications:
      - name: my-transformer-model
        import_path: my-transformer-model:deployment
        runtime_env:
          working_dir: "https://my-git-repo.com/main.zip"             ❷
  rayClusterConfig:                                                   ❸
    rayVersion: %VERSION%                                             ❹
    headGroupSpec:
      ...
      template:
        spec:
          containers:
          - name: ray-head
            image: rayproject/ray-ml:%VERSION%
            ports:
            ...                                                       ❺
            - containerPort: 8000
              name: serve
    workerGroupSpecs:
    - replicas: 1
      groupName: gpu-group
      template:
        spec:
          containers:
          - name: ray-worker
            image: rayproject/ray-ml:%VERSION%
          tolerations:                                                ❻
            - key: "ray.io/node-type"
              operator: "Equal"
              value: "worker"
              effect: "NoSchedule"
```

❶ This field is where all the configuration of the Ray Serve application is

❷ The code of the application is downloaded from working_dir location

❸ This section of the spec is to configure head and worker nodes of Ray Cluster

❹ The version of Ray should specified here and in the images to use

❺ The head node exposes multiple components in addition to the serving aspect, like dashboard or client

❻ As in some previous examples, it is possible to configure Tolerations and Taints to match GPU requirements

From a platform/Kubernetes perspective Ray is definitely less familiar in terms of API and management compared to KServe, but at the same time it enables data scientists and python developers to have a full control over deployment. This flexibility brings a lot of value especially when you need to configure a more complex serving topology, like distributed serving or training on multiple hosts.

Lessons learned

We are still at the beginning of the journey with Generative AI and Kubernetes, in this chapter we covered the main components necessary to execute a LLM on Kubernetes.

We started loading a LLM programmatically using Hugging Face Transformer library, then introduced the concept of Model Server and finally the Model Server Controller to manage the integration and the lifecycle with Kubernetes.

The provided examples are not intended to be comprehensive, each Model Server has a different configuration and supports different models/optimizations but the approach is equivalent so you should be able to adapt them to your needs. These differences are even bigger if we compare KServe and KubeRay.

This field is evolving fast and new projects are created every day, we decided to focus on the principles that are more mature and adopted. We introduced multiple technologies in this chapter but in the following chapters we will focus on a single implementation per component. In the context of Model Server to serve LLM we will default to vLLM in the rest of the book given that it is the project that has most community adoption while in the context of Model Server Controller we will default to KServe because it is built natively in Kubernetes while Ray has an approach that is alternative and partially in competition with Kubernetes.

CHAPTER 3

Model Data

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

As we have already seen in the previous chapters, one of the largest challenges is to bring in the LLM data into a Kubernetes cluster so that it can be leveraged by runtimes.

The main portion of those models consists mainly of the parameters of the model and can be extremly large. Table 3-1 contains the number of parameters and size of some more prominent available models that you can run yourself. There are many more, but from this selection you can already see a wide range of variations, ranging from large models that are likely impractical for on-demand use to more lightweight models that can be run on your own cluster and easily downloaded when needed.

Table 3-1. Sample models and their sizes

Name	Vendor	Parameters	Size
Llama 3.1 405B	Meta	405 billion	~750 GB
OPT-175B	Meta	175 billion	~350 GB

Name	Vendor	Parameters	Size
Vicuna	LMSYS	33 billion	~66 GB
Orca	Microsoft	13 billion	~26 GB
Granite 13B	IBM	13 billion	~26 GB
Falcon 2	TII	11 billion	~22 GB
LLaMA 3	Meta	8 billion	~16 GB
Mistral 7B	Mistral	7 billion	~14 GB

Even smaller models can pose significant challenges for Kubernetes administrators when it comes to efficient handling and management within a cluster. Understanding how to store and organize these large datasets effectively is critical for a successful LLM operation.

In this chapter, we will explore how to manage data-heavy artifacts efficiently within a Kubernetes cluster. Most of the time, ML models can be treated as opaque boxes, accessed by the inference services described in Chapter 2. However, understanding the package formats used to distribute these models is still valuable for proper integration. "Model Data Storage Formats" on page 48 provides an overview of the most important LLM storage formats.

Another critical aspect of operating LLMs is discovering where to find and how to retrieve model data. The concept of *Model Registries*, discussed in "Model registry" on page 60, offers a practical solution for model discovery and access.

Finally, the models must be downloaded into the cluster to be usable. "Accessing model data in Kubernetes" on page 71 outlines Kubernetes-native methods for efficiently fetching and accessing model data.

Model Data Storage Formats

The first thing we notice when working with LLMs is their massive size, measured in billions of parameters. However, models shared on platforms like Hugging Face contain more than just the raw weight parameters. They also include metadata and, in some cases, the model's architecture, which defines how the neural network layers and transformers are wired together.

For operators, such distributed models often feel like black boxes. Yet, understanding in which format they are stored is critical because not every packaged model can run with every runtime described in Chapter 2. Some formats are highly flexible and can be operated by multiple runtimes, while others are closely tied to specific runtime platforms.

At a high level, model storage formats can be grouped into two categories:

Weights-Only Formats

These formats store only the learned parameters (weights and biases) of a neural network. The architecture, hyperparameters, and metadata are excluded, so the runtime must already know how to reconstruct the network before applying the weights.

Self-Contained Formats

Self-contained formats store both the weights and the model architecture, along with hyperparameters and other metadata. They allow the model to be loaded and run without requiring prior knowledge of the network structure, making them easier to deploy as standalone artifacts.

The boundary between both categories is gradual. Some formats that seem self-contained may still require external components, such as tokenizer files for language models.

For LLMs, the trend is moving towards such *mostly self-contained* formats like GGUF and Safetensors. These formats simplify distribution but remain tightly coupled to specialized runtimes. True runtime independence where a model could be loaded and run in any compatible environment, regardless of its training framework, remains a work in progress. The CNAI Model specification described in "CNAI Model Specification" on page 77 is a standardization attempt in this direction, where model data is packed in OCI container images.

In an ideal world, much like OCI containers abstract application internals, model storage formats would draw a clear boundary between model data (produced by data scientists) and model execution (managed by MLOps/DevOps engineers in production). However, today's landscape prioritizes getting models operational quickly rather than standardizing runtime compatibility. As the field matures, expect stronger separation between model creation and deployment concerns.

Weight-Only Formats

Weight-only model formats store the numerical parameters (weights and biases) of a trained neural network without including the model's architecture or preprocessing components. These formats are commonly used during the development and experimentation phases, where flexibility and minimal overhead are more important.

Since weight-only formats lack architectural details, the runtime must have prior knowledge of the network structure to correctly reconstruct the model and apply the stored weights. These formats are tightly coupled to their respective machine learning frameworks.

Some common weight-only formats used for LLMs and other AI models:

PyTorch State Dict (`.pt`, `.pth`)
: PyTorch's native format for serializing weight tensors using the `state_dict` dictionary. It is widely used for LLMs such as LLama, GPT, and BLOOM during development and fine-tuning stages.

TensorFlow Checkpoints (`.ckpt`)
: A format primarily used in TensorFlow's ecosystem for storing model weights. While it was historically used for models like BERT, its relevance for modern LLMs has declined as PyTorch gained some dominance in the GenAI space.

NumPy Arrays (`.npy`, `.npz`)
: NumPy's native serialization format for numerical arrays. While still useful for storing smaller models or individual weight matrices, it lacks the structure and metadata needed for modern LLM deployments.

These formats primarily store raw tensor data with minimal metadata, making them highly compact but dependent on external runtime code.

As illustrated in Figure 3-1, a model stored in a weight-only format requires the same network architecture to be reconstructed during inference. The training architecture must be manually replicated in the inference environment, ensuring both sides can correctly interpret the stored weight tensors.

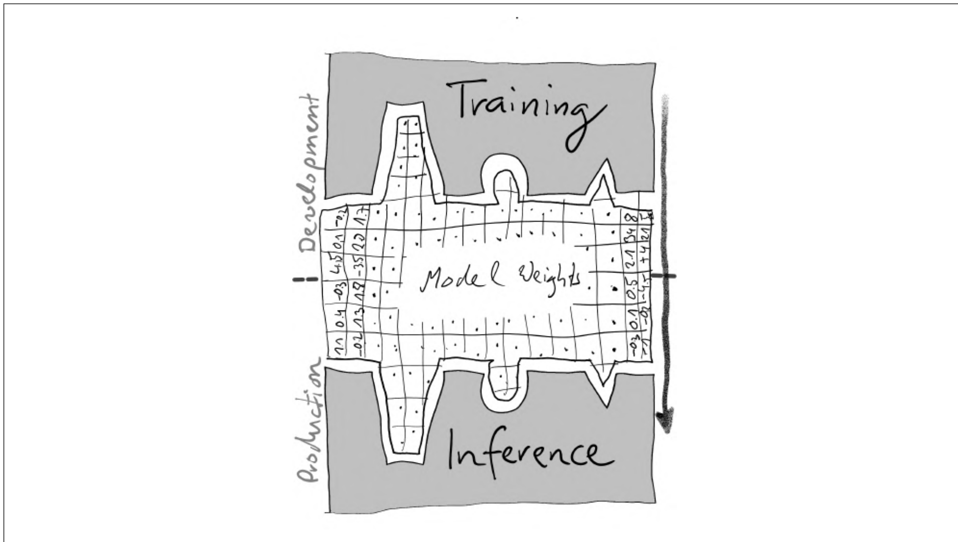


Figure 3-1. Example of a model stored in a weight-only format

While weight-only storage formats are well suited during the development and experimentation phase, they are very closely coupled to the ML code that evaluates those parameters.

Self-contained Formats

A better fit for production deployments are models stored and distributed in *self-contained* formats, which bundle more than just the raw weights. These formats include critical metadata and structural information, making models easier to share and run across multiple runtime environments without requiring the original codebase used during training.

Self-contained models can carry the following information:

- **Weights and biases**: The numerical parameters of the neural network, which make up the bulk of the model size.
- **Model architecture**: Either as a reference to a well-known architecture or described explicitly as a connected graph of layers.
- **Tokenizer and Vocabulary Data**: Often included in language models to preprocess text before inference.
- **Hyperparameters**: Information like learning rate, batch size, and number of epochs used during training.
- **Other Metadata**: Descriptive information such as model origin, authorship, and additional context for model discovery and reproducibility.

Some self-contained formats also support **pre- and post-processing** scripts for transforming inputs before inference and converting outputs into a usable form afterward.

Figure 3-2 illustrates a model stored in a self-contained format, where all components are bundled together, enabling runtime independence from the original training code.

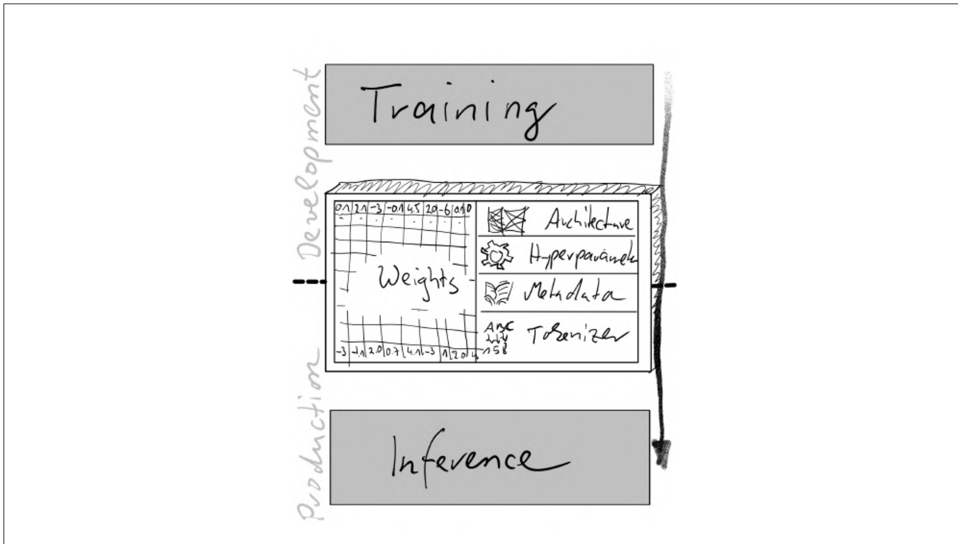


Figure 3-2. Example of a self-contained model where the runtime is independent of the training code.

While fully self-contained formats aim to encapsulate everything needed for inference, *mostly self-contained* formats still rely on some external components and runtime dependencies. These formats may bundle the model weights and partial metadata but often omit critical components like the tokenizer or detailed model architecture. As a result, they remain tied to specific inference runtimes or frameworks that "understand" how to interpret the stored data correctly.

For example, Safetensors includes model weights and limited metadata but typically requires a separate tokenizer and model definition during inference. GGUF and GGML store quantized weights and some runtime metadata but expect a compatible runtime like `llama.cpp` for execution.

In current LLM practice, fully self-contained models do not yet exist. No widely used format today includes all components required for inference like the tokenizer, vocabulary data, and the complete model architecture in a single artifact. As a result, even the formats often described as "self-contained" are better categorized as *mostly self-contained* because they still depend on external files or runtime knowledge to some degree.

Common *mostly self-contained* formats for LLMs include:

Safetensors (`.safetensors`)
: A mostly self-contained format designed for secure and efficient weight storage, frequently used for LLMs on platforms like Hugging Face. While it improves safety and performance over standard PyTorch weight files, tokenizer informa-

tion (e.g., `tokenizer.json`) and model architecture definitions are not embedded, requiring additional files or runtime knowledge to fully reconstruct the model during inference. See "Safetensors" on page 55 for more details.

GGUF/GGML (`.gguf`, `.ggml`)
: Specialized self-contained formats optimized for CPU-based inference with quantized weights. They include the model's weights and basic architecture metadata but remain closely tied to runtimes like llama.cpp and vLLM, which are designed to efficiently handle the quantized structures. GGUF can store the tokenizer data (like vocabulary data and special tokens) but is still connected to specific runtimes like llama.cpp or vLLM. See "GGUF and GGML" on page 57 for more information about GGUF.

ONNX (`.onnx`)
: A versatile, self-contained format for model interoperability. Often described as self-contained, ONNX stores the model's weights, architecture, and metadata but lacks critical components like the tokenizer and vocabulary data, which are essential for LLMs. This makes it mostly self-contained, requiring additional files for complete language model inference. See "ONNX" on page 54 for more details.

TensorFlow SavedModel
: A fully self-contained, directory-based format that stores weights, architecture, and auxiliary files. While common in TensorFlow ecosystems, it is rarely used for modern LLMs.

HuggingFace Transformers
: The "Hugging Face Transformers format" is best described as a packaging convention rather than a standalone model format. It organizes models into a directory containing multiple files essential for running language models. This convention typically includes the model's weights stored in formats like Safetensors (`.safetensors`) or PyTorch's `state_dict` (`.bin`) along with two key files: `tokenizer.json` and `config.json`. These files play a crucial role in ensuring the model can process input data and apply the correct architecture during inference.

tokenizer.json and config.json

The `tokenizer.json` and `config.json` files are critical components for running LLMs effectively in the Hugging Face ecosystem and beyond. The `tokenizer.json` file stores the tokenization rules and vocabulary mapping for converting raw text into token IDs. It defines how input text is split into tokens, using techniques like Byte Pair Encoding (BPE), and includes special tokens used for padding, start-of-sequence, and end-of-sequence markers. The `config.json` file describes the model architecture and hyperparameters, containing information such as the number of layers, attention

heads, hidden sizes, and feed-forward dimensions. It often specifies the model type (e.g., `llama`) and influences how the runtime reconstructs the model graph. Together, these files ensure the model can preprocess input correctly (`tokenizer.json`) and build the required network structure (`config.json`). Without them, the runtime cannot properly tokenize input text or load the model architecture for inference.

These files have become de facto standards in the machine learning community, extending their utility beyond the Hugging Face ecosystem. Frameworks and tools outside of Hugging Face often adopt these conventions for model interoperability and consistency.

While there isn't a formal schema specification publicly available for these files, their consistent structure and widespread adoption have established them as reliable standards for model configuration.

As we have seen, most current model formats for LLMs fall into the category of *mostly self-contained*, often omitting key components such as tokenizers, vocabulary data, and preprocessing logic. Despite these gaps, some formats have gained significant traction due to their balance between portability and efficiency. The most commonly used for LLM deployments today are Safetensors and GGUF/GGML, both optimized for efficient weight storage with metadata. While ONNX is less frequently used for LLMs, it serves as a useful reference for a more fully self-contained format, though it would require additional elements like tokenizer definitions to be truly complete. In the following sections, we will explore ONNX, SafeTensors, and GGUF/GGML in more detail.

ONNX

The Open Neural Network Exchange (ONNX), co-developed by Microsoft and Facebook in 2017, was designed as a framework-independent format for representing machine learning models. It aimed to standardize how models are shared between tools, allowing developers to train a model in one framework and deploy it in another without requiring framework-specific conversions.

ONNX models are stored in a single `.onnx` file using Protocol Buffers (protobuf) for compactness and platform neutrality. Each file contains the model's computational graph, which defines the network's structure and the flow of data, the learned model parameters such as weights and biases, and metadata describing input and output specifications, operator sets, and versioning details. This structure makes ONNX a promising example of a self-contained format, as it combines architecture, weights, and operational metadata in a single artifact.

However, ONNX falls short for LLMs because it lacks essential components such as tokenizers, vocabulary data, and preprocessing logic. For tasks like natural language

generation, this missing information makes it necessary to supply additional files alongside the `.onnx` model. Without these components, an ONNX model alone cannot transform raw text into tokenized inputs, limiting its suitability for modern LLM deployments. This gap prevents it from being fully self-contained in the context of language models.

ONNX's broad support across runtimes like ONNX Runtime, TensorRT, OpenVINO, and Triton Inference Server makes it highly portable, but compatibility depends on the set of operations a model uses. Each runtime supports a defined operator set (`op set`), which specifies the available operations a model can use. If a model relies on operations outside a runtime's supported set, it may fail to load unless extended with plugins or custom runtime extensions. This challenge further complicates its adoption for complex architectures like those used in LLMs, where tokenization and text preprocessing steps are integral parts of the model's functionality.

Despite these limitations, ONNX provides a conceptual blueprint for what a fully self-contained model format for LLMs could look like. If expanded with richer metadata and native support for tokenizer definitions, it could offer a more complete solution for the LLM use case. For now, however, ONNX remains better suited for models in domains like computer vision, where preprocessing is often simpler and less tightly coupled with the model.

Next, we'll explore Safetensors, a format more commonly used for LLM deployment today, offering optimized weight handling and some degree of metadata inclusion.

Safetensors

Safetensors, developed by Hugging Face in 2021, is a modern model serialization format designed to securely store and share machine learning model weights while addressing security vulnerabilities and performance limitations of earlier formats like PyTorch's `.pt` and `pickle`. The pickle format, often used in PyTorch, can execute arbitrary Python code when deserializing models, posing significant security risks when sharing models. In contrast, Safetensors prevents code execution vulnerabilities by focusing strictly on storing tensor data, making it a safer and more efficient choice for model serialization.

Safetensors files follow a simple yet efficient structure, as shown in Figure 3-3.

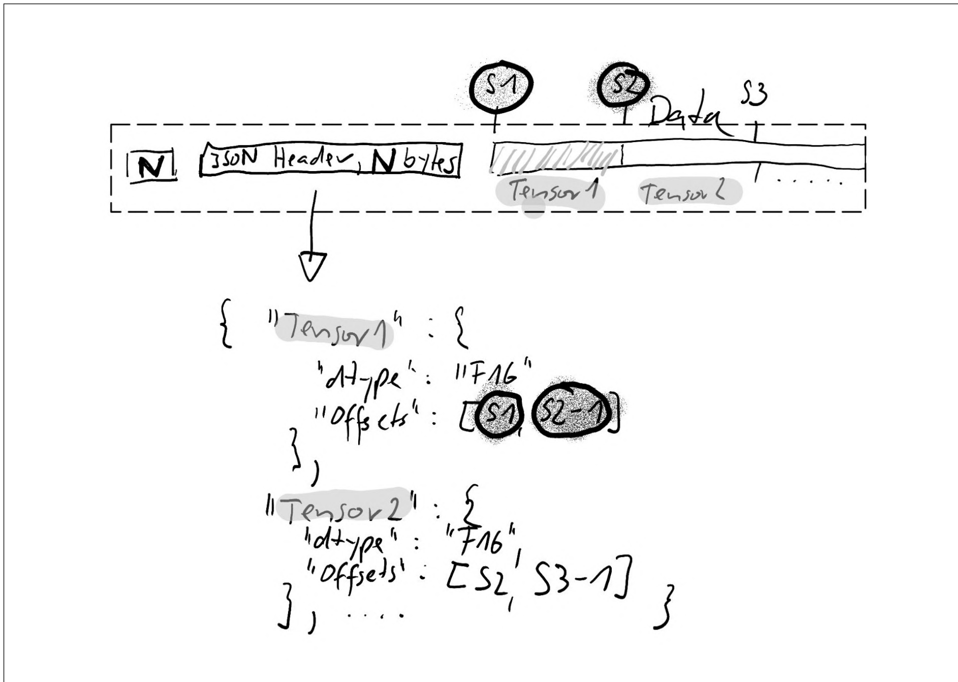


Figure 3-3. Internal structure of a Safetensors model.

Each `.safetensors` file begins with a header containing metadata, including a serialized JSON object describing each tensor stored in the file. The header includes details such as the tensor's data type, shape, and the byte offsets where the tensor data resides within the file. This structure allows for zero-copy loading, where tensor data can be directly mapped to memory without unnecessary CPU overhead, improving inference speed, especially when working with LLMs.

Safetensors supports sharding, which allows large models to be split across multiple smaller files. Each shard contains a portion of the model's tensors and is accompanied by an index file (e.g., `model.safetensors.index.json`). The index file maps the names of tensors in the different layers to their respective shard files. For example, Llama 4.1 405B is released with 30 safetensor files named like `model-0000x-of-00030.safetensors` and accompanied by a `model.safetensors.index.json` file that looks like Example 3-1.

Example 3-1. Example of a model.safetensors.index.json for sharded safe tensor files

```
{
  "metadata": {
    "total_size": 141107412992
  },
```

```
  "weight_map": {
    "lm_head.weight": "model-00030-of-00030.safetensors",
    "model.embed_tokens.weight": "model-00001-of-00030.safetensors",
    "model.layers.0.input_layernorm.weight": "model-00001-of-00030.safetensors",
    "model.layers.0.mlp.down_proj.weight": "model-00001-of-00030.safetensors",
    "model.layers.0.mlp.gate_proj.weight": "model-00001-of-00030.safetensors",
    "model.layers.0.mlp.up_proj.weight": "model-00001-of-00030.safetensors",
   ...
    "model.layers.1.input_layernorm.weight": "model-00002-of-00030.safetensors",
    "model.layers.1.mlp.down_proj.weight": "model-00002-of-00030.safetensors",
    "model.layers.1.mlp.gate_proj.weight": "model-00001-of-00030.safetensors",
    "model.layers.1.mlp.up_proj.weight": "model-00002-of-00030.safetensors",
   ...
  }
}
```

Sharding is particularly useful for extremely large models where a single file might be impractical due to storage limitations. It also enables parallel loading, as different shards can be fetched and processed concurrently.

While Safetensors improves the safety and performance of model weight storage, it still falls into the category of mostly self-contained formats rather than fully self-contained. The primary limitation is that tokenizer information and model architecture definitions are not included within the `.safetensors` file itself. Essential files like `tokenizer.json` and `config.json` must be supplied separately for language model inference, which is a key reason why it remains tightly coupled to the Hugging Face Transformers ecosystem that offers this extra meta data.

The format's structure and focus on secure serialization have made it increasingly popular, especially for LLM storage and sharing. Safetensors is now the default weight format for many large-scale models distributed on Hugging Face.

Next, we will explore GGUF, a more specialized format for LLMs which is optimized for CPU-based inference and designed for efficient deployment of LLMs.

GGUF and GGML

The GGUF (GPT-Generated Unified Format) and its predecessor GGML (GPT-Generated Model Language) are specialized formats developed for optimizing the storage and execution of LLMs in resource-constrained environments such as CPUs and edge devices. Originating from the llama.cpp project led by Georgi Gerganov, both formats focus on efficient inference with minimal hardware requirements. While GGML was an important first step, GGUF represents a significant refinement, addressing many of its predecessor's limitations.

A defining feature of GGUF and GGML is their focus on quantization, a technique that reduces the precision of model weights from floating-point values to lower-bit representations such as 8-bit, 4-bit, or even 2-bit integers. By lowering precision, both

the memory footprint and computational overhead are significantly reduced, allowing models to run effectively without dedicated GPUs while maintaining acceptable inference accuracy.

GGML was initially created as a lightweight single-file format for sharing and running LLMs on CPUs. However, as models grew more complex, GGML struggled with flexibility. Users often needed to adjust quantization parameters and normalization settings manually, leading to compatibility issues with newer models and inference runtimes. GGUF, introduced in August 2023, was designed to address these challenges while expanding the format's capabilities. It offers a richer metadata structure, improved support for model architecture definitions, and better handling of quantized weights while retaining the lightweight characteristics that made GGML popular for CPU inference.

A key improvement in GGUF is its focus on backward compatibility. As LLMs evolve and their architectures become more complex, maintaining compatibility with existing tools can be challenging. GGUF's modular design allows newer models to retain compatibility with older runtime versions, provided the core components remain unchanged. This helps prevent the need for frequent format conversions when updating models. The backward compatibility design also minimizes the impact of transitions between versions. When GGUF is updated to support new features, existing models remain functional without requiring conversion.

Unlike ONNX, which was designed as a general-purpose format for a wide range of machine learning tasks, GGUF is specialized for LLM inference. It focuses on efficient CPU execution and is widely supported by runtimes like llama.cpp, vLLM, and other CPU-optimized frameworks.

When compared to Safetensors, GGUF attempts to bundle more metadata directly within the model file itself, including basic tokenizer information and runtime metadata. While Safetensors focuses primarily on weight storage with minimal metadata and relies on external files for tokenizer definitions and model configurations, GGUF stores token mappings and model parameters in a single file. GGUF still depends on specific external runtimes for complete inference, keeping it in the category of mostly self-contained formats.

A GGUF file consists of a structured binary layout, beginning with a magic number and version field to identify the file type, followed by a section containing quantized tensor data stored with byte offsets for efficient access. The metadata section describes the model's architecture, quantization type, and token mappings. The tensor information block defines the data type, shape, and memory locations for each tensor stored in the file. This single-file design is particularly beneficial in Kubernetes environments, where consistent, self-contained artifacts simplify orchestration and scaling. Figure 3-4 illustrates the structure of a GGUF file.

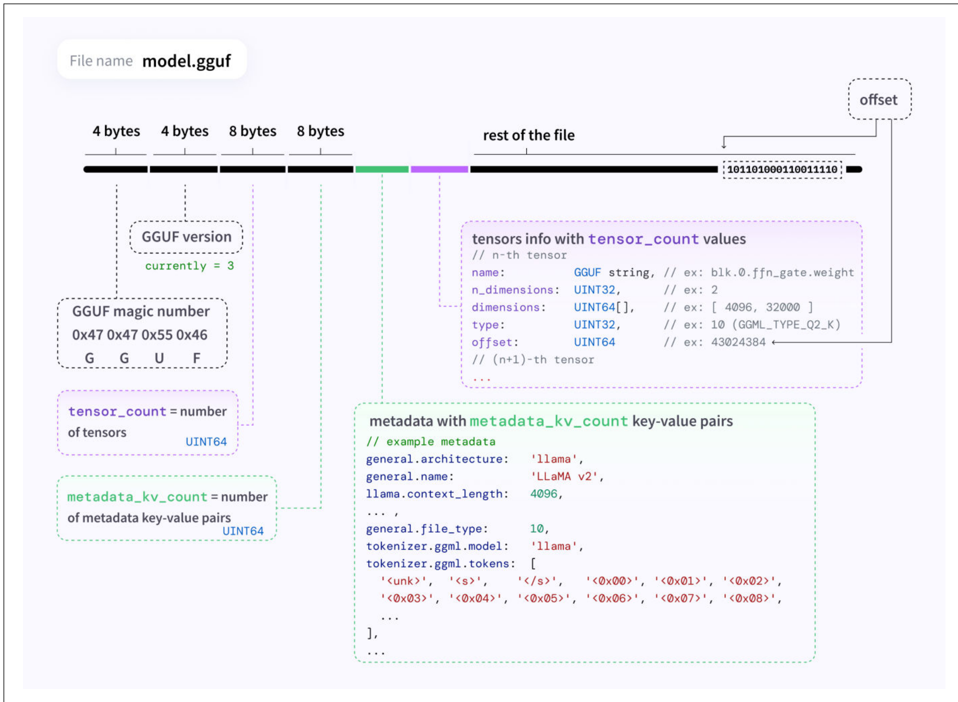


Figure 3-4. Internal structure of a GGUF file.

GGUF represents a leap forward for deploying LLMs efficiently, especially on hardware that lacks high-end GPUs. Its focus on quantization, self-contained design, and backward compatibility addresses many pain points of earlier formats.

What's next ?

While ONNX stands out as a self-contained format for general machine learning models and GGUF offers a specialized, self-contained solution for LLMs, both formats reveal important gaps in model portability.

ONNX provides a structured way to package models but lacks critical components like tokenizers for LLMs, while GGUF includes basic tokenizer metadata but remains tightly coupled to specific runtimes like `llama.cpp`. A future goal should be to achieve true model portability, where models can be distributed and executed as self-contained artifacts, much like how Docker revolutionized the deployment of arbitrary software workloads across diverse environments.

Reaching this level of portability would require broader standardization across both the model file structure and the runtimes capable of executing them. Ideally, a model stored in a standardized format could be loaded by any compliant runtime without manual adjustments for tokenization, quantization, or architecture specifics. Such a

shift would empower a more diverse set of tools and frameworks, reducing lock-in to specific ecosystems while making model distribution as seamless as containerized applications.

The landscape of LLM development is still evolving rapidly. New architectures, optimization techniques, and runtime improvements emerge frequently, each introducing specialized configurations and breaking the idea of a universal, all-encompassing standard. Until the dust settles and the field matures, mostly self-contained formats like GGUF and Safetensors will likely remain the most practical choices for balancing performance, compatibility, and flexibility. True standardization, much like OCI's success, will require the convergence of both runtime capabilities and model representation standards, a milestone that is still some distance away.

Understanding the structure and formats of model files helps in selecting the right tools and runtimes, but ultimately, a LLM is just a collection of files, whether fully self-contained or spread across multiple artifacts. Managing these files effectively in Kubernetes environments requires a way to index, discover, and organize them, which is the role of a model registry which we talk about next.

Model registry

A model registry provides a central system for managing models, track versions, governance and store metadata about ML artifacts. It plays a crucial role in the machine learning lifecycle by bridging the gap between model experimentation and production deployment. Serving as both a discovery mechanism and a collaboration platform, a model registry simplifies how models are tracked, verified, and deployed at scale.

Unlike public registries, most model registries are deployed as local services within a cluster. These registries are not exposed to the outside of the cluster. They primarily manage model metadata rather than storing the actual model weights or artifacts. Instead, they reference external object stores like AWS S3 buckets where the actual model data resides. This separation of metadata and model storage ensures greater flexibility in managing large models while keeping metadata easily accessible within the cluster.

By providing a structured and secure interface for managing models and their metadata, model registries become a critical tool for operationalizing machine learning at scale, especially in dynamic environments like Kubernetes.

A model registry stands at the intersection of the responsibilities of data scientists and MLOps engineers. For data scientists, it supports creating and tracking changes during model experimentation, verifying performance and metric tracking, packaging artifacts for reproducibility, and releasing validated models to production. For MLOps engineers, the model registry facilitates deploying approved models with

associated metadata while also supporting ongoing monitoring of deployed models for performance, drift, and necessary retraining, though this level of observability is considered an advanced feature beyond the core functionality of a model registry.

The following list outlines the core features that define a model registry, providing essential capabilities for both public and local use cases:

Metadata Management
: Store information about model accuracy, dataset lineage, performance benchmarks, and other critical metadata.

Model Discovery and Search
: Search and retrieve models based on metadata such as architecture, hyperparameters, training datasets, and performance metrics.

Version Control
: Track multiple versions of models. Versioning enables comparison of different model iterations and rollback if necessary.

Lifecycle Management
: Manage model stages such as experimentation, staging, production, and retirement. This feature is especially critical as part of continuous development workflows.

Access Control
: Provide fine-grained permissions for model visibility and usage, ensuring secure collaboration across teams.

Auditing and Compliance
: Maintain a record of model usage, approvals, and changes to ensure regulatory compliance and reproducibility.

Data Pipelines
: Integrate into CI/CD workflows, automating tasks like model validation, artifact packaging, and production rollout.

To provide a clearer understanding of how these features are implemented in real-world tools, we will examine four prominent model registries: Hugging Face Model Hub, MLflow Model Registry, Kubeflow Model Registry, and OCI Registries.

Hugging Face Model Hub

The Hugging Face Model Hub is the canonical platform for discovering and sharing open-source machine learning models, including LLMs. As of early 2025, it hosts over 1.2 million models in general and more than 160,000 LLMs in specific, all publicly available. Much like GitHub serves as the primary hub for open-source

software development, Hugging Face has established itself as the leading platform for open-source ML models.

Each model entry in the catalog is accompanied by a *Model Card*. A Model Card provides a standardized summary of a machine learning model's key characteristics, including its intended use case, training datasets, performance benchmarks, and limitations. It often contains links to the datasets used for training, evaluation metrics, and licensing information. Users can also try out models interactively using the built-in inference widget, which enables quick testing of the model directly from the web interface without requiring local setup (Figure 3-5).

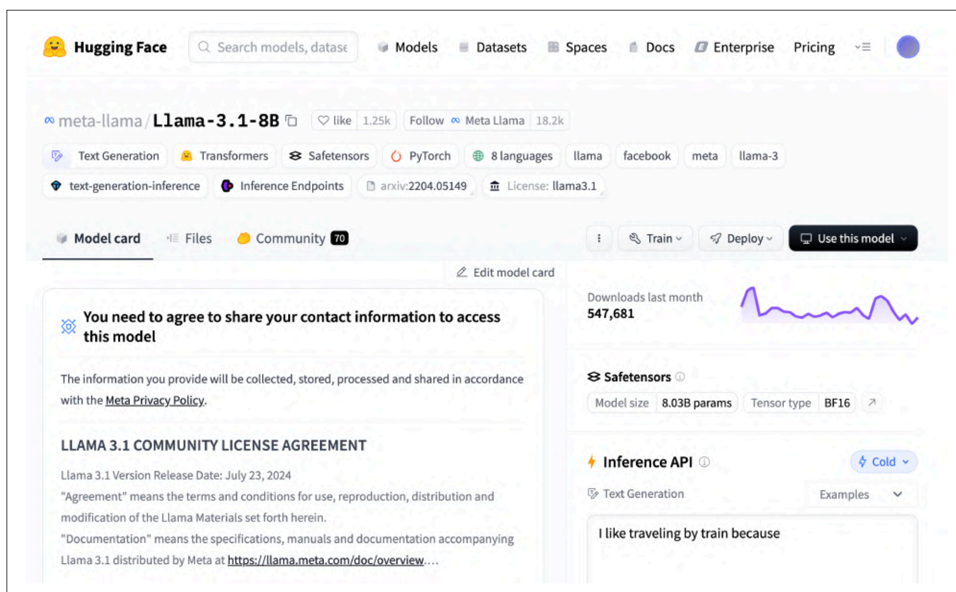


Figure 3-5. Hugging Face Model Card for Llama 3.1

In addition to the web interface, Hugging Face also offers a REST API for programmatic access to its repository. This allows developers to query models, retrieve metadata, and integrate models directly into automated workflows and pipelines. The API simplifies tasks such as discovering the latest version of a model or filtering models based on specific criteria.

While the Hugging Face Hub is perfect for manual discovery and collaboration, it may become limiting in fully automated workflows where model versions need to be programmatically tracked and managed. For such scenarios, a dedicated model registry becomes essential to ensure version control, traceability, and tighter integration into production pipelines.

MLflow Model Registry

MLflow is a comprehensive toolset designed to manage the machine learning lifecycle, including experiment tracking, model packaging, and model registry functionalities.

MLflow was created by Databricks in 2018 to address the challenges of managing machine learning experiments and model artifacts consistently across teams and environments. Since its release as an open-source project, MLflow has become widely adopted in the data science community for its simplicity and integration capabilities.

The central element of MLflow is the *Tracking Server*, which acts as the main hub for managing and storing all experiment metadata, metrics, and model artifacts. It provides an interface where data scientists can log results, compare runs, and organize their models and expose them in the model registry. A rich set of visualization allows following the change of performance data and different hyperparamaters. The models themselves are stored in the simplest case locally on the file-system. For production setups, MLflow supports pushing model artifacts to external storage systems like AWS S3 or downloading directly from the Hugging Face Hub. MLflow manages references to these storage locations through artifact URIs stored in the registry's metadata.

The MLflow Model Registry is a part of this Tracking Server, providing a centralized repository for versioning, tracking, and managing machine learning models. It allows data scientists to register models with rich metadata, including version history and performance metrics. Figure 3-6 shows the Web UI of the Model Registry.

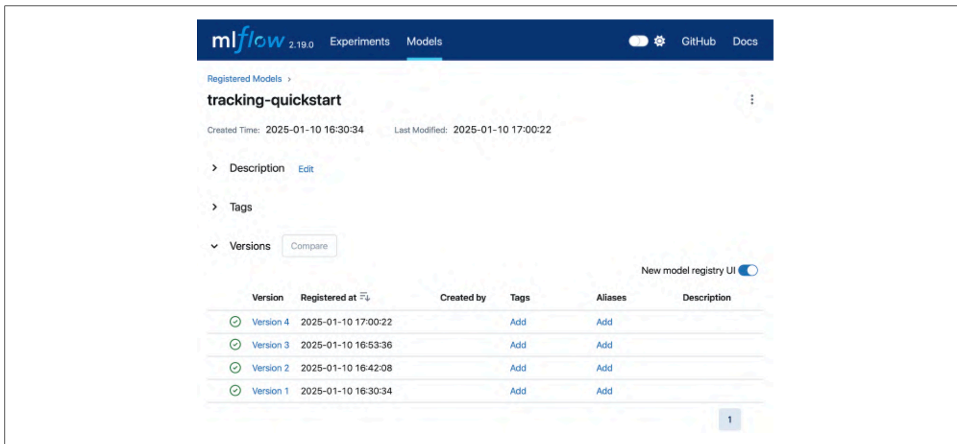


Figure 3-6. MLflow Registry UI

Most of the time however data scientists interact with the MLflow model registry programmatically like in Example 3-2.

Example 3-2. Programmatically logging and registering models with MLflow

```
mlflow.set_tracking_uri(uri="http://127.0.0.1:8000") ❶
mlflow.set_experiment("MLflow Demo")                  ❷

params = {                                            ❸
    "solver": "lbfgs",
    "multi_class": "auto",
    "max_iter": 2500,
}

with mlflow.start_run():
    mlflow.log_params(params)                         ❹
    model_info = mlflow.sklearn.log_model(            ❺
        sk_model=model,
        artifact_path="my_model",
        input_example=X_train,
        registered_model_name="my-model",
    )
```

❶ Set tracking server uri for logging

❷ Create a new MLflow Experiment

❸ Model hyperparameters

❹ Log those hyperparameters

❺ Log the model itself at the tracking server. The definition of `model` and `X_train` are not show here

For MLOps engineers, MLflow provides a REST-API that you can leverage for discovery of models. Example 3-3 shows how you can fetch the details of a given model.

Example 3-3. Searching for and listing of models via MLflows REST API

```
$ curl http://localhost:8000/api/2.0/mlflow/registered-models/search   ❶

{
  "registered_models": [
    {
      "name": "my-model",
      "creation_timestamp": 1736523034148,
      "last_updated_timestamp": 1736524822538,
      "latest_versions": [
        {
          "name": "my-model",
          "version": "4",                                              ❷
          "creation_timestamp": 1736524822538,
```

```
        "last_updated_timestamp": 1736524822538,
        "current_stage": "None",
        "description": "",
        "source": "mlflow-artifacts:/84948067/f0dd25483e/artifacts/my_model",   ❸
        "run_id": "f0dd25483e234400b7",
        "status": "READY",
        "run_link": ""
      }
    ]
  }
 ]
}
```

❶ Accessing an MLflow server running on the local machine

❷ Model in the registry are versioned

❸ Reference to the model artifacts, stored locally here

MLflow provides CLI tools that interact with the Model Server as shown in Example 3-4. An interesting option here is to create a self-contained OCI container image that you can push to an OCI registry for later usage in an Kubernetes cluster. However, this feature is not optimized for large download volumes that need to be stored locally, so it is not very well suited for LLMs. You can push such image to an OCI registry for later usage in a Kubernetes cluster. We describe how OCI registries can be used for model data in "OCI Registry" on page 70.

Example 3-4. Creating a self-contained OCI container image with MLflow and Podman

```
$ mlflow models generate-dockerfile \         ❶
  -m mlflow-artifacts:/84948067/f0dd25483e/artifacts/my_model
... INFO mlflow.models.cli: Generating Dockerfile for model mlflow-artifacts:
    .../artifacts/my_model
... INFO mlflow.models.flavor_backend_registry: Selected backend
    for flavor 'python_function'
... INFO mlflow.models.cli: Generated Dockerfile in directory mlflow-dockerfile

$ cd mlflow-dockerfile
$ podman build -t my_model .                  ❷
STEP 1/12: FROM python:3.13.1-slim
STEP 2/12: RUN apt-get -y update && apt-get install -y --no-install-recommends nginx
....
Successfully tagged localhost/my_model:latest
a828556afe0d53d4728d872aa51fe07eaa1d4ef4faedb5a788bac9a7a7651e73
```

❶ Use the `mlflow` CLI to generate a Dockerfile that describe how to build an image with MLflow and the model data included.

❷ Use `podman` to create an OCI image named `my_model`. Alternatively, you can also use Docker for building the image.

While MLflow was not initially built with Kubernetes in mind, it can be deployed effectively on a Kubernetes platform. The standard approach is deploying it as a web service using tools like Helm charts, where a PostgreSQL database often serves as the backend for storing metadata. MLflow does not introduce native Kubernetes CRDs, which means its integration with Kubernetes requires additional automation for tasks such as scaling and dynamic model serving.

MLflow, while feature-rich, is not perfectly suited for running LLMs. Its metadata management and artifact handling are well-suited for traditional ML use cases, but LLMs often require specialized handling due to their size and complexity. MLflow has introduced some support for working with large models through the Transformers flavor, including memory-efficient and storage-efficient logging techniques. For example, MLflow provides options for logging large models without loading them into memory and referencing external models hosted on the Hugging Face Hub instead of storing weights locally. However, these approaches can create challenges in production environments, such as the risk of losing access to external repositories or insufficient caching mechanisms for repeated large model retrievals. As a result, MLflow's artifact storage and model handling techniques, though improving, remain less suited for the specific demands of LLM management at scale. For example, downloading large models repeatedly from a registry can become inefficient, and MLflow's current artifact storage approach is not optimized for such high-volume data handling.

In summary, MLflow is primarily focused on the data science side of the ML lifecycle, providing a rich feature set for tracking data science experiements. It's biggest advantage is that it is very accessible and can be easily installed on local machines. The challenge is to connect it to production-ready platforms like Kubernetes for delivering large models.

These gaps are addressed by tools like Kubeflow, which extend the concept of a model registry with deeper Kubernetes integration and additional observability features.

Kubeflow Model Registry

Kubeflow is a Kubernetes-native platform designed to simplify the entire machine learning lifecycle, including model training, serving, and model registry management. Initially developed by Google, Kubeflow is now an open-source project under the Cloud Native Computing Foundation (CNCF).

It consist of these loosely connected components:

Kubeflow Dashboard
: A central dashboard is our hub which connects the authenticated web interfaces of Kubeflow and other ecosystem components.

Kubeflow Notebooks
: Component for running web-based development environments like Jupyter Notebooks inside your Kubernetes cluster by running them inside Pods. No local installation is needed.

Kubeflow Pipelines
: Kubeflow Pipelines (KFP) is a platform for building then deploying portable and scalable machine learning workflows using Kubernetes.

Katib
: Katib is a Kubernetes-native project for automated machine learning (AutoML) with support for hyperparameter tuning, early stopping and neural architecture search.

Model Training
: Kubeflow Trainer is a unified interface for model training and fine-tuning on Kubernetes. It runs scalable and distributed training jobs for popular frameworks like PyTorch or TensorFlow.

Model Serving
: KServe (previously KFServing) solves production model serving on Kubernetes. It started in Kubeflow but has been moved to a separate CNCF project. We cover KServe in detail in "KServe" on page 38.

Model Registry
: Index and catalog for ML models. The registry is the central hub within the Kubeflow ecosystem. The rest of this section will focus on this registry.

Figure 3-7 gives an overview of how the Model registry interacts with the other parts of Kubeflow.

At its core, Kubeflow takes advantage of Kubernetes principles, with all tasks, including model registration and training, defined as containerized workloads. Unlike MLflow, which is a more flexible experiment tracking and model management tool, Kubeflow offers deeper Kubernetes integration through CRDs and native controllers for each ML lifecycle component.

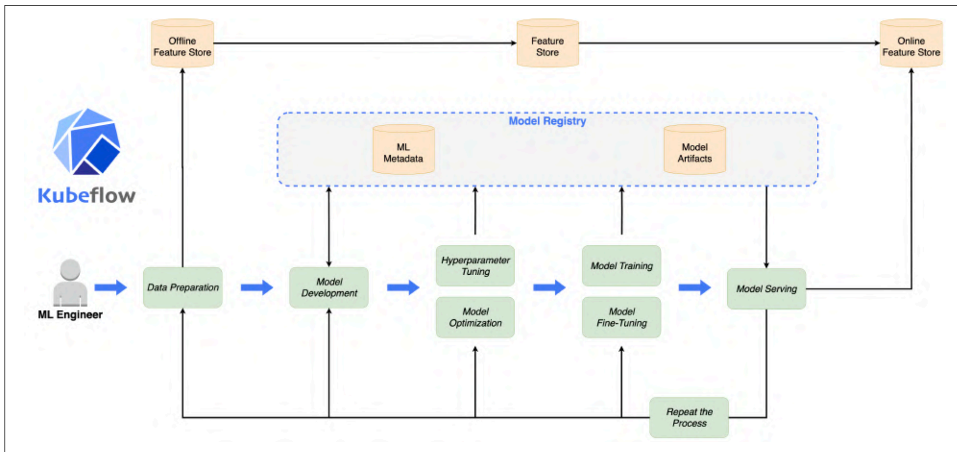


Figure 3-7. Kubeflow architecture and how it interacts with its Model registry

The Kubeflow Model Registry serves as a central repository for managing machine learning models, their versions, and related metadata. It substantially simplifies the transition from experimentation to production deployments.

At its core, the registry utilizes Google's ML Metadata (MLMD) as its backend for metadata storage and management. This integration ensures a structured, scalable approach to storing model lineage, metrics, and parameters. With MLMD, the Kubeflow Model Registry can standardize metadata, enable version control, and offer interoperability across Kubeflow components. This allows for robust tracking of model versions and the reuse of metadata for deployment or pipeline triggers.

The registry relies on external dependencies such as MySQL for metadata storage, with a persistent volume required for durability. This needs to be taken into account when operating the registry in production setups. It exposes REST APIs and a Python SDK for interaction.

To use the registry you need to register a model first, along with its meta data. Example 3-5 shows how you can do this from within a Python program or a Jupyter notebook.

Example 3-5. Register a model at the Kubeflow Model Registry

```
from model_registry import ModelRegistry

registry = ModelRegistry(                         ❶
    server_address="http://model-registry-service.kubeflow.svc.cluster.local",
    port=8080,
    author="your name",
    is_secure=False
)
```

```
rm = registry.register_model(                          ❷
    "iris",
    "gs://kfserving-examples/models/sklearn/1.0/model",
    model_format_name="sklearn",
    model_format_version="1",
    version="v1",
    description="Iris scikit-learn model",
    metadata={
        "accuracy": 3.14,
        "license": "BSD 3-Clause License",
    }
```

❶ Create a proxy to the Model registry running in the cluster.

❷ Register a model with meta data and reference to the location of the model data

When a model is registered at the registry, you can easily access this via a Python library call. You can also access the model via an REST API call directly to the service, as shown in Example 3-6.

Example 3-6. Run a curl command from within the cluster to query the cluster-internal model registry

```
kubectl run -it --rm curl --image=curl --restart=Never \        ❶
   http://model-registry-service.kubeflow.svc.cluster.local/...
```

❶ Run a curl inside the cluster to query the model registry

You can also access the Kubeflow Model registry with a KServe InferenceService in order to initialize the InferenceService with the model data that the registry points to. See Example 3-7 for an example how to do this.

Example 3-7. Example of an InferenceService that accesses the model data via a Kubeflow registry.

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: iris-model
spec:
  predictor:
    model:
      storageUri: "model-registry://iris/v1"     ❶
      modelFormat:
        name: "sklearn"                          ❷
        version: "1"
```

❶ Reference to the model id and version

❷ Format that specifies the runtime to use

OCI Registry

An OCI (Open Container Initiative) registry is a standard mechanism for storing and distributing container images, commonly used in Kubernetes environments. Familiar services like Docker Hub and Quay.io have made it easy for Kubernetes users to store and manage images without running a registry themselves. Some Kubernetes distributions, such as Red Hat OpenShift, even include a built-in OCI registry.

What is OCI ?

The Open Container Initiative (OCI) standardizes how containerized applications and artifacts are managed. Founded in 2015 by Docker and others under the Linux Foundation, OCI ensures interoperability and vendor neutrality in container technologies. It evolved from Docker's proprietary format to avoid lock-in, in favour of an open, extensible ecosystem.

While OCI began with container images, it now supports diverse artifacts like Helm charts and generative AI models through its OCI Artifacts specification. This makes registries highly versatile for modern workloads.

An OCI registry can store more than just container images. With the introduction of OCI 1.1, the specification expanded to support OCI artifacts, a generalization of the original image format. OCI artifacts allow storing arbitrary data types, making an OCI registry suitable for hosting machine learning models, including LLMs. This means the registry can manage the entire model file rather than merely referencing external storage.

OCI registries provide versioning, immutability, and efficient distribution mechanisms that fit well with LLM hosting. Compared to MLflow and Kubeflow registries, which primarily store model metadata and references to external storage, an OCI registry focus to store the full model data itself.

LLM model images are examples of "passive data images". They are not meant to be executed but serve as immutable packages of model weights and configurations for inference runtimes. You can easily create such a data image by cloning a Hugging Face repository as shown in Example 3-8.

Example 3-8. Dockerfile for creating a container image that holds a model

```
FROM alpine/git
RUN git lfs install \
```

```
 && git clone https://huggingface.co/Qwen/Qwen2.5-0.5B-Instruct /models
ENTRYPOINT sh
```

This Dockerfile can be be used directly with `podman` or `docker` as shown in Example 3-9 to create a self-contained OCI image files that has all files needed to run the model.

Example 3-9. Build and push a model file with podman

```
$ podman build -f Dockerfile.model -t quay.io/rhuss/qwen2.5-0.5b-instruct . ❶

STEP 1/3: FROM alpine/git
Trying to pull docker.io/alpine/git:latest...
Getting image source signatures
...
Writing manifest to image destination
STEP 2/3: RUN git lfs install
       && git clone https://huggingface.co/Qwen/Qwen2.5-0.5B-Instruct
       && ln -s /git/Qwen2.5-0.5B-Instruct /models
Git LFS initialized.
Cloning into 'Qwen2.5-0.5B-Instruct'...
--> b437a8f78e49
STEP 3/3: ENTRYPOINT sh
COMMIT quay.io/rhuss/qwen2.5-0.5b-instruct
--> f680df7c975f
Successfully tagged quay.io/rhuss/qwen2.5-0.5b-instruct:latest
f680df7c975f6bfc806783574003c2b17872e9bf767944380f

$ podman push quay.io/rhuss/qwen2.5-0.5b-instruct:latest ❷
```

❶ Build model image. It will clone the full repo from Hugging Face Hub and might take a bit.

❷ Push to the registry from where you can access it from the Kubernetes cluster.

By leveraging OCI registries, you can store, version, and distribute LLM models efficiently within Kubernetes-native infrastructure, integrating smoothly into MLOps pipelines and declarative workflows.

Accessing model data in Kubernetes

Now that we have seen the various model formats and solution how to register them for tracking and ease of discovery, let's go into the details and learn how we can access the model data from within a Kubernetes cluster.

Chapter 2 described several ways how GenAI models can be served on Kubernetes. They all require the models to be downloaded in some way. For all runtimes described in Chapter 2 there exist similar methods for getting hold of the model

data, but for demonstration purpose let's stick to KServe as the prototypical example here.

In the simplest case, the storage location is specified in an InferenceService resource as shown in Example 3-10 by leveraging a `storageUri` that points to the model's data location.

Example 3-10. InferenceService picking up model data from a S3 storage

```
apiVersion: "serving.kserve.io/v1beta1"
kind: "InferenceService"
metadata:
  name: "mnist"
spec:
  predictor:
    serviceAccountName: sa                        ❶
    tensorflow:                                   ❷
      storageUri: "s3://kserve-examples/mnist"   ❸
```

❶ Kubernetes ServiceAccount that is associated with a Secret that holds the AWS authentication credentials.

❷ The runtime to use, TensorFlow in this example.

❸ Reference to a S3 bucket that holds the model data files.

The schema of this URI defines which backend and where the model data is stored. Each schema triggers a so called *storage initializer* which eventually translates into a runtime's Pod init-container. You can create and deploy your own storage initializers with KServe's ClusterStorageContainer resource. As shown in Example 3-11, in this resource you specify a reference to an image holding the custom storage initializer and a list of URL schemas that should trigger that storage initializer. URL's that match these schemas can then be used as `storageUri` specification in an InferenceService.

Example 3-11. ClusterStorageContainer resource that adds a `model-registry://` schema for `storageUri` usage

```
kind: ClusterStorageContainer
metadata:
  name: model-registry-storage
spec:
  container:
    name: storage-initializer
    image: kubeflow/model-registry-storage-initializer ❶
  supportedUriFormats:
    - prefix: model-registry://                        ❷
```

❶ Reference to OCI image for executing the initializer logic.

❷ Register URL schema `model-registry` so that it can be used in an InferenceService.

The storage initializer is run as an init-container before the model runtimes start and its only purpose is to make the model data available for the serving runtime.

Init Containers and Sidecars

Init Containers and Sidecars are powerful Kubernetes patterns for enhancing Pod behavior. Init containers run first and perform one-time setup tasks, such as populating a shared volume with data needed by the main container. Sidecars, on the other hand, run alongside the main container, often providing auxiliary functionality like logging, data processing, or cross-container data sharing. Together, these patterns enable a flexible and modular design for Pods. For more insights, check out the Init Container and Sidecar patterns described Kubernetes Patterns.

Table 3-2 shows the storage initializers that KServe supports out of the box.

Table 3-2. KService storage initializers

Schema	Description	Example
gs	Download from Google Cloud Storage	`gs://kfserving-examples/models/sklearn/1.0/model`
s3	Download from an AWS S3 bucket	`s3://kserve-examples/mnist`
https	Download model data with HTTP	`https://awesome-llms.com/models/llama-3.2-7b`
hdfs, webhdfs	Access files from an Hadoop Distributed File System	`hdfs://path/to/model`
pvc	Copy model data from an PersistentVolume reference by the given PersistentVolumeClaim	`pvc://${PVC_NAME}/export`
oci	Pull OCI image with model data and access it directly via a modelcar, see "Modelcars" on page 77.	`oci://quay.io/rhuss/kserving-example-sklearn:1.0`
model-registry	Access a model registered at the Kubeflow Registry. See "Kubeflow Model Registry" on page 66 for more details about this type of model registry.	`model-registry://iris/v1`
hf	Download directly from Hugging Face Hub	`hf://meta-llama/Llama-2-7b-chat-hf`

A common pattern in Kubernetes is sharing data among containers using dedicated node-local volumes. Most of the storage initializers from Table 3-2 download the model data into a node-local directory that then is shared and mounted by a LLM runtime so that it can access it directly. For this purpose, Kubernetes provides the `emptyDir` volume type, that is initialized as an empty directory and mountable by all containers within the same Pod — whether they are init containers running first or application containers running after the init containers. The model serving runtime then mounts this volume to access the prepared data. For more details and variations of this pattern, refer to the *Immutable Configuration* pattern in Kubernetes Patterns.

Let's see how we can use OCI images for transfering and storing model data, and how we can leverage this for smoothely accessing the model parameters with the LLM runtimes.

OCI image for storing model data

It was in 2013 when Docker invented a clever layered format for storing container blueprints. The original and still prevalent usage for those images is to store all the binaries and files that make up a Linux operating system, beside the kernel. It is a layered format so that people can create *base images* which can be reused for different specialized images that e.g. contain the applications that are to be run in a container. Layers are shared when multiple containers are running that refer to the same layers.

In addition to the read-only layers of an image, Docker uses a union filesystem that adds a read-write layer on top of the image layer stack, so that different container instances can still share the same underlying operating system files. One key benefit of this schema is that the read-only layers can be cached individually, which makes working with OCI images very efficient as only changed layers need to be distributed.

We don't go into much details about the concrete format here as many aspects are not relevant when we store model data in such layers. Important for the moment is, that you can share layers and that an OCI image is built up hierarchically, i.e. layers are stacked. This stacking matches nicely for model composition techniques like finetuning with LoRA adapters on top of foundational models. These foundational models, stored in base images, can be shared when running on the cluster nodes, which makes it very efficient to run multiple specialized fine-tuned models.

Figure 3-8 shows how such images are composed. At the end all layers are packed into a tar archive that is stored at an OCI registry.

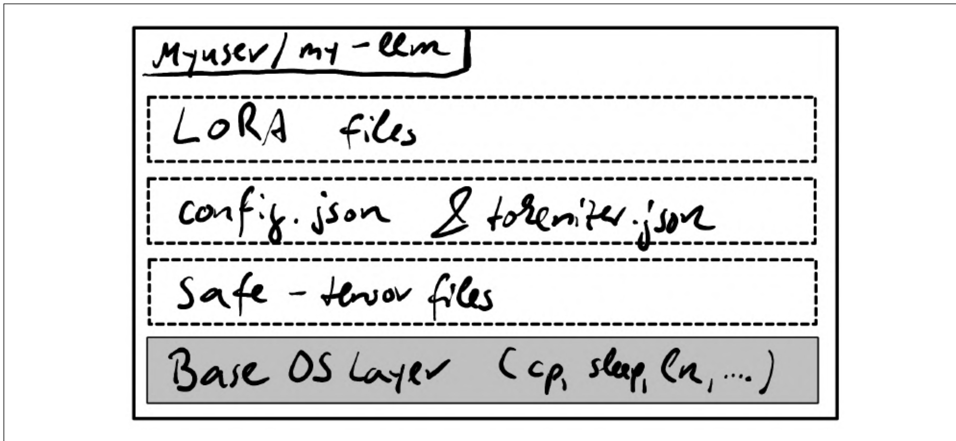


Figure 3-8. OCI Image consists of multiple directory-layer

Docker's success eventually lead to a standardization of the image specification by the OCI. A full ecosystem of supporting tools from registries for hosting OCI images to CLI tooling like `skopeo` or `oras` for inspecting and managing OCI images has emerged over time. By putting LLMs into OCI images piggy backs on this existing landscape and benefits automatically from the existing work that has been done in this area.

In "DIY - Do It Yourself" on page 35 we've seen how to deploy a LLM model with a vanilla Kubernetes Deployment resource. In Example 2-8 the model data is downloaded on the fly from the Hugging Face Hub, but we could also initialize the model data directly from an OCI container image. Example 3-12 shows a similar Deployment, but this time we are introducing an `emptyDir` volume for sharing the model data.

Example 3-12. Deployment with an init-container that copies over model data to a local emptyDir volume

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: vllm
spec:
  replicas: 1
  template:
    spec:
      initContainers:
      - name: copy-model-data
        image: quay.io/meta-llama/meta-llama-3.2-8b  ❶
        command:
        - "sh"
```

```yaml
    - "-c"
    - "cp -a /models/. /mnt/models"                  ❷
    volumeMounts:
    - name: models
      mountPath: /mnt/models                         ❸
  containers:
  - name: vllm
    image: vllm/vllm-openai:latest
    args:
    - "--served-model-name",
    - "meta-llama/Meta-Llama-3-8B",
    - "--model",
      "/mnt/models"                                  ❹
    volumeMounts:
    - name: models                                   ❺
      mountPath: /mnt/models
  volumes:
    - name: models                                   ❻
      emptyDir: {}
```

❶ (Fictive) OCI image holding the model data for Llama 3.2 in a directory `/models`.

❷ Copy over the data from the image directory `/models` to the mounted `/mnt/models` directory that is backed by an `emptyDir` volume. This might take some time depending on the size of the model to copy.

❸ Mount the emptyDir volume to the `/mnt/models` in the init container.

❹ Run vLLM so that it access the model stored in `/mnt/models`.

❺ Mount the shared directory on `/mnt/models` in the application container to access the data copied by the init-container.

❻ Volume declaration for an empty node-local directory .

The technique demonstrated in Example 3-12 shows how model data is typically initialized for a deployed model, independent if its downloaded from a S3 bucket or extracted from an OCI image. Beside downloading the data from some source it involves an expensive copy step that is performed everytime a runtime Pod is started.

The following two sections demonstrates how this copying over of gigabyte-sized amounts of data can be avoided by directly accessing the data that is contained in an OCI model data image.

> ### CNAI Model Specification
>
> The Cloud Native AI (CNAI) Model Specification is an emerging effort to extend the OCI image specification for packaging and distributing AI models. It targets an expansion of the OCI standard to support AI model artifacts, including model weights, metadata, and configurations. The goal is to standardize model storage and management, ensuring better compatibility across different runtime environments. By leveraging OCI's extensible architecture, it aims to simplify model deployment and sharing. This initiative complements OCI's image volume mount capabilities described later in "Modelcars" on page 77 and "OCI Image Volume Mounts" on page 84. The definition of new annotation types is also part of the specification. While still in early stages, the initiative has already gained interest from the community and is expected to seek CNCF adoption in 2025. Its success will lead to a more unified approach to operationalizing AI workloads in cloud-native environments.

Modelcars

As we have seen in Example 3-12, you can easily access model's stored in OCI images. However this way of copying all the model data into an intermediate storage has some drawbacks.

Wouldn't it be awesome if we could just directly access the model data stored in an OCI image, withouth copying it first ?

This would not only speed up the initialization for serving runtimes, but also is more mindful about local node space. An image needs to be downloaded only once, but can be used simultaneously by many Pods. Also, for an LLM model that can benefit from the layered nature of OCI images (like LoRA finetuned models), the overall storage space that is needed for specialized models that are based on the same foundational model is reduced. The image layers of the foundation model can be shared among the specialized models, reducing the required disk space considerably.

Kubernetes has long lacked support for this use case. Although the feature request was already recorded more than ten years ago in GitHub issue 831, it was not considered for implementation for many years.

However, things have changed with the advent of LLMs and the desire to ship model data in OCI images. Beginning with Kubernetes 1.31 you can use now image volume mounts directly in your Pod specs (when you enable this experimental feature). It might take some time though until image volume mounts move out of the experimental stage and are considered to be stable.

We talk about OCI image volume mounts in detail later, but let's look at how KServe uses a trick to achieve the same behaviour for older Kubernetes versions. You might consider jumping directly to "OCI Image Volume Mounts" on page 84 if you already

can leverage OCI volume mounts, since modelcars can be considered as a temporary solution that you can use today. OCI image volume support will support everything that modelcars provide, but is a much cleaner and standardized technique. You should use OCI image volumes whenever you can, and rely on modelcars if this is not yet possible.

Let's see how modelcars can be used in KServe today. Example 3-13 shows how a modelcar can be configured in KServe. The model data that is stored in the image that is referenced with an `oci://` URL will be directly accessed *without prior copying into a volume* like demonstrated in Example 3-12. Modelcars can speed up the startup of a model runtime considerably, expecially when working with a large data set.

Example 3-13. Inference service that uses model data from an OCI image

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: "sklearn-iris-oci"
spec:
  predictor:
    model:
      modelFormat:
        name: sklearn
      storageUri: "oci://rhuss/kserving-example-sklearn:1.0" ❶
```

❶ OCI registry and repository of image holding the model data

The remaining part of this section is a deep dive in the technical architecture and implementation of modelcars. The level of detail is higher than the most of the rest of the book, so feel free to skip this section and jump directly to "OCI Image Volume Mounts" on page 84. However, we feel that the pattern behind this technique proves to be useful in other scenarios when you have to deal with large amount of data, so we'll keep it here for some technical fun and eductional purposes.

The Kubernetes' Pod specification supports a relatively unknown property called `shareProcessNamespace`. By default, containers that are started on behalf of a Pod can not see each other. I.e. when you do a `ps aux` inside a container, you will only see the processes that are started by this container. This is great to keep containers isolated. When you set `shareProcessNamespace` to `true`, the container "sees" other processes of other containers. You can also access the *filesystem* from all containers via the `/proc` filesystem.

Example Example 3-14 shows how this cross-container filesystem access can be tested.

Example 3-14. Accessing an other container's root file system

```
$ cat spns.yaml

apiVersion: v1
kind: Pod                                ❶
metadata:
  name: spns
spec:
  containers:
  - image: docker.io/httpd
    name: httpd
  - image: docker.io/busybox
    name: busybox
    command: ["sleep", "infinity"]
  shareProcessNamespace: false

$ kubectl apply -f spns.yaml

# Jump into the busybox container
$ kubectl exec -it spns -c busybox -- sh

$$ ps
PID   USER     TIME  COMMAND
    1 root      0:00 sleep infinity   ❷
    7 root      0:00 sh
   14 root      0:00 ps aux

$$ ls -d /proc [0-9]*
/proc/1  /proc/7

# Root filesystem of PID 1
$$ ls /proc/1/root/                      ❸
bin     dev     etc     home    lib     lib64
proc    root    run     sys     tmp     usr     var

# Jump out of the container again
$$ exit

# Change `shareProcessNamespace` from false to true
$ sed  's/false/true/' spns.yml | kubectl apply --force -f -

# Jump into busybox container like before
$ kubectl exec -it spns -c busybox -- sh\
$$ ps

PID   USER     TIME  COMMAND
    1 root      0:00 /pause           ❹
```

```
  7 root      0:00 httpd -DFOREGROUND
 15 www-data  0:00 httpd -DFOREGROUND
 16 www-data  0:00 httpd -DFOREGROUND
 17 www-data  0:00 httpd -DFOREGROUND
 99 root      0:00 sleep infinity
126 root      0:00 sh
132 root      0:00 ps

# Show data from the others container
$$ head -3 /proc/7/root/usr/local/apache2/conf/httpd.conf   ❺
#
# This is the main Apache HTTP server configuration file.  It contains the
# configuration directives that give the server its instructions.
```

❶ Simple Pod with two containers: An Apache HTTP server and a busybox that sleeps forever to keep the container running. No process namespace sharing is enabled here.

❷ Only the processes from the container's process namespace are visible. Note that the specified command has PID 1 when process namespace isolation is enabled.

❸ Root filesystem of process PID 1 (which is the same as `ls /`)

❹ When process namespace sharing is enabled, the PIDs from the other containers can be seen, too.

❺ Via the proc filesystem, a file specific to the httpd-container can be accessed from the busybox container.

Accessing other processes' filesystem is only possible when Unix permissions allow. Ideally the processes from all containers use the same UID, so that cross-container filesystem access should not be an issue. However, depending on your cluster setup additional mechanisms like SELinux might affect the abilitiy to access another container's filesystem, even when using the same UID or using UID 0 for the containers.

This technique to cross-share the containers' filesystems is universal to Kubernetes and can be used for any deployed workload, regardless if you have deployed the runtime yourself or via an add-on platform.

Although it's not necessary to understand what happens behind the scene, its enlightening how KServe implements direct image mounting. The technique is independent of KServe and can also be used in other contexts where access to large datasets stored in OCI images is required.

Figure 3-9 shows the components and structure of a modelcar in KServe.

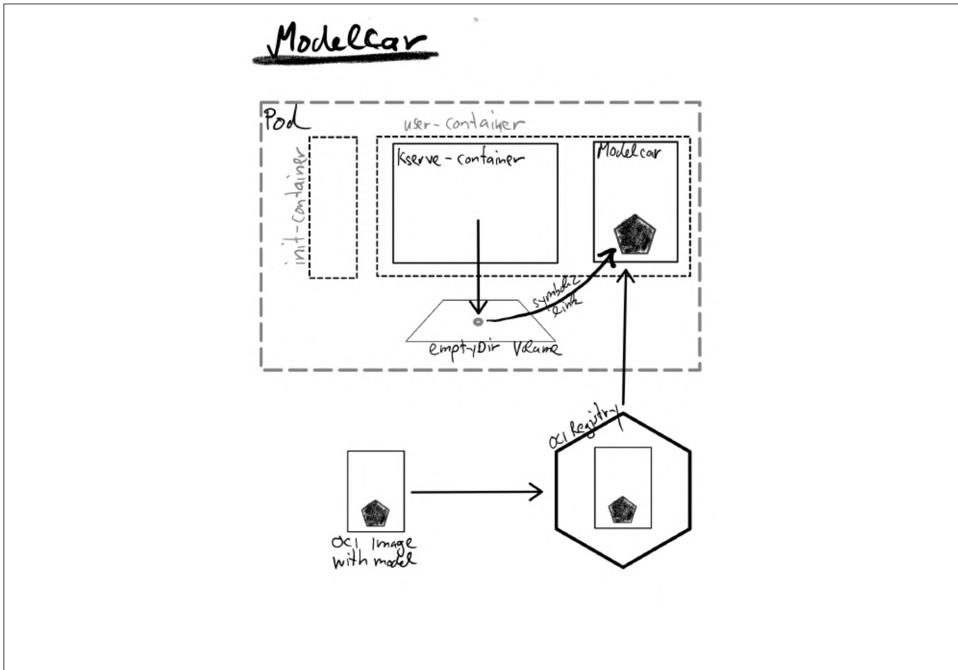


Figure 3-9. Modelcar components

The serving runtime and the modelcar container are starting in parallel. During the startup, the modelcar creates a symbolic link from its file system to a shared `emptyDir` volume accessible by by both containers. Then, the modelcar goes into an infinite sleep, to keep the container alive.

This linking operation is part of the modelcar's startup command and requires minimal resources — less than 10MB of memory to maintain idle status. It's important to emphasize that no data is copied over; just a symbolic link is created to allow the serving runtime container to find the model data under a fixed location (e.g. `/mnt/models`).

Example 3-15 shows how what a Pod definition looks like, that results on behalf of the creation of an `InferenceService`. The important part here is the creation of the link and the mount of the shared `emptyDir` volume to hold the symbolic link to follow for cross-container access.

Example 3-15. Pod with a Modelcar sidecar that creates a symbolic link in a shared volume to point to its own image data via `/proc/<id>/root`.

```
apiVersion: "serving.kserve.io/v1beta1"
apiVersion: v1
kind: Pod
metadata:
  name: sklearn-iris-oci-predictor-00001-deployment-7fd9c7fc67-dzdsz
  namespace: default
spec:
  shareProcessNamespace: true
  containers:
  - name: kserve-container
    image: kserve/sklearnserver                                    ❶
    args:
    - --model_name=sklearn-iris-oci
    - --model_dir=/mnt/models
    volumeMounts:
    - mountPath: /mnt                                              ❷
      name: kserve-provision-location
  - name: modelcar
    image: rhuss/kserving-example-sklearn:1.0                      ❸
    args:
    - sh
    - -c
    - ln -s /proc/$$$$/root/models /mnt/models && sleep infinity   ❹
    volumeMounts:
    - mountPath: /mnt
      name: kserve-provision-location
  volumes:
  - name: kserve-provision-location                                ❺
    emptyDir: {}
```

❶ Serving runtime that executes on the model from the modelcar.

❷ Mounting the shared local directory on `/mnt` so that the model can be accessed from `/mnt/models`.

❸ Modelcar image that holds the model data.

❹ Creates a symbolic link `/mnt/models` that points into the modelcar's own root filesystem, accessible via the proc filesystem. `$$$$` get replaced in YAML to `$$` which is the special shell variable that holds the modelcar's shell process id. After the link is created the modelcar sleeps indefinitely to keep the container alive.

❺ Declaration of the shared empty dir volume that is referenced in the container declaration for the serving runtime and the modelcar.

While this technique proved to be very valuable for optimizing the initialization of LLMs there are also a handful of drawbacks of this Modelcar approach:

Startup Order
: Serving runtime typically assume that the model data is already present when those runtimes start up. However, in the case of a modelcar, the modelcar container and the runtime container are started in parallel, which can lead to the situation that the model is not yet available when the runtime starts. Despite the fact that modelcar containers are starting very quickly, it will be slower in startup when the modelcar image still needs to be pulled from an OCI image registry. This can be mitigated by usind the Kubernetes sidecar support that is available since Kubernetes 1.28 as optional features, so that the runtime only starts when the modelcar is initialized. For setups where sidecars are not enabled you still can minimize the risk of a race condition by pre-pulling the modelcar image in an init-container so that it is ensured that when the modelcar sidecar starts, that the modecal OCI image is already present at the cluster node.

Security
: Enabling `shareProcessNamespace` allows the access to the process names space and filesystems of **all** containers defined for a Pod. This is especially important to remember when there are also other sidecars included. A prominent example is the service mesh Istio that uses sidecars to provide its functionality. Istio sidecars assume that they are fully isolated, so they do not create any precautions to hide sensitive information like the access configuration to their upstream Istio daemon. As shown in this security report the lack of additional encryption of the local Istio configuration can be easily exploited. Therefore its important to understand the consequences when using tools and platforms that perform sidecar injections like Istio or Knative.

Non-Uniform Startup Times
: Depending on whether the model OCI image has been already loaded in the Kubernetes' node OCI runtime, the actual serving runtime can either start quickly or it might take several minutes until a potentially large model OCI image is downloaded from a registry. To make the startup times more predictable, which is important especially in scale-to-zero scenarios, optimization techniques like image prefetching can be leveraged.

Multi-Arch Support
: Modelcars require an active process to keep the sidecar alive. This process is specific to a certain CPU architecture, so if you want to use modelcar images in a multi-architecture setup, then you need to create copies of modelcars, one for each supported CPU architecture. Those images are containing the same ML model leading to a waste of resources.

All those drawbacks can be overcome by *real* OCI image volume mounts. Luckily, Kubernetes 1.31 introduce OCI image sources for volumes as an experimental feature. It will still take some time until this mount type will be generally available, in the meantime Modelcars are a good bridging technology with a smooth upgrade path until OCI image volume mounts eventually arrive for everyone.

OCI Image Volume Mounts

Starting with Kubernetes 1.31, Pods can directly mount OCI container images as volumes without the need to copy model data first. This feature provides an efficient way to access large model artifacts stored in OCI images, reducing both initialization time and storage overhead.

The benefit of direct image mounts over the Modelcar approach (see "Modelcars" on page 77) is that it avoids the need for symbolic links or process namespace sharing. Instead, model data can be directly read from the image layers as a mounted volume, benefiting from the underlying OCI image layer cache.

As of early 2025 this feature is still experimental, you need to enable it explicitly via the feature gate `ImageVolume` to enable in the configuration of the Kubernetes API server.

Example 3-16 shows how to use an OCI image volume mount to serve a model directly with vLLM.

Example 3-16. Pod serving a locally mounted LLM via vLLM

```
apiVersion: v1
kind: Pod
metadata:
  name: llm-server
spec:
  containers:
  - name: main
    image: vllm/vllm-openai:latest   ❶
    args:
      - "--served-model-name"
      - "meta-llama/Meta-Llama-3-8B"
      - "--model"                    ❷
      - "/mnt/models"
    volumeMounts:                    ❸
      - name: model-volume
        mountPath: /mnt/models
  volumes:
  - name: model-volume
    image:                           ❹
      reference: quay.io/meta-llama/meta-llama-3.2-8b
      pullPolicy: IfNotPresent       ❺
```

❶ Runtime image for serving the model, `vLLM` in this case.

❷ Specify an absolute path to the mounted model as startup argument for `vLLM`.

❸ Mount content of OCI image into `/mnt/models`.

❹ `image:` is the volume type for an OCI image to mount. The usual pull semantics for images applies: If no `pullPolicy` is provided, always pull the image if tag or tag `latest` is specified. Otherwise Kubernetes pulls only if the image is not present at the node.

❺ Pull policy can be also specified explicitly.

While this alpha feature simplifies large model deployments, it still has limitations:

- Only works with `cri-o` as of Kubernetes 1.31.
- Feature gates must be explicitly enabled.
- No support for writeable layers; volumes are read-only.
- No support for OCI artifacts, only OCI images are supported

The community is actively working on these limitations. This feature will eventually become the preferred method for serving LLMs on Kubernetes, replacing the Modelcar approach as it matures. In the meantime, modelcars are a reliable approach for direct access to model data stored in an OCI image.

More Information

- Safetensors vs GGUF
- ONNX
- Safetensors
- GGUF / GGML
- MLflow vs Kubeflow
- OCI Image Volume Mounts

CHAPTER 4

Model Observability

> ### A Note for Early Release Readers
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

In Chapter 2 we learned how to deploy a LLM in Kubernetes starting from scratch with a simple coding example. The full stack included a Model Server, vLLM, to optimize the execution of the model and a Model Server Controller, KServe, to manage the integration with Kubernetes and the lifecycle of the deployment.

Then in Chapter 3 we focused on LLM model data, with the complexity and the options that are available today to manage the size of similar models. We are getting closer and closer to a full production setup where the LLM workload is fully managed and automated so that it can be executed side by side to the other workloads (i.e. traditional applications) all managed by Kubernetes.

Kubernetes is a very powerful and complex platform to orchestrate container execution with a clear declarative API and with promise to self heal the workload thanks to controllers and reconcile loop in an eventually consistent way. Everyone that has Kubernetes experiences knows that this approach doesn't replace proper observability and monitoring of the workload so that it is possible to quickly react when something

cannot be solved automatically. As you can imagine this principle applies to LLM too, it is critical to monitor a Model Server but, given the nature of LLM, it is not equivalent to monitor traditional applications.

LLMs are quite different in terms of how they produce workload, definitely different compared to a traditional microservice with few endpoints where the workload is mainly driven by number of requests and speed of query on data. LLMs are different even compared to traditional ML!

In this chapter we will see why they are different, which aspect of the execution is important to be monitored and the corresponding available metrics.

Understanding LLM

The goal of this book is not to explain the theory behind LLMs or the details of their implementation. However, it is necessary to cover some aspects of a model's processing logic to better understand what needs to be monitored and which metrics are available. The focus of this section is the inference pipeline, detailing the steps performed from the moment a request reaches our vLLM endpoint to the generation of the output.

As mentioned previously, Large Language Models are a subset of the models under the Generative AI category, they are based on Transformer architecture and used to process text (natural language) to perform a number of different tasks. An example of a task is to produce a summary of a longer text, another is to ask the model to answer user questions or to classify some data. The Transformer architecture describes an *encoding* phase and a *decoding* phase, this has been used to create three different classes of models: *encoder only* models, *encoder-decoder* models and *decoder-only* models.

In general encoder models are popular for learning embeddings used in classification tasks (i.e. Google BERT or Meta RoBERTa), encoder-decoder models are a good fit for generative tasks like translation / summarization where input and output are strongly connected (i.e. Google Flan-T5), and decoder-only models are used for generative tasks like Q&A (i.e. OpenAI GPT-1/2/3).

In practice, today the majority of models adopted for text generation are decoder-only and they are able to perform translation/summarization pretty well without the need of the encoder step. We will focus on decoder-only models, but vLLM can also serve encoder-decoder models, and the inference pipeline described here is analogous.

From a practical perspective, an LLM is a complex neural network that processes and generates numbers rather than text. Therefore it requires a conversion layer to make it more usable. A way to perform this conversion is to create a huge vocabulary of

all possible words and use the index of this vocabulary as an integer representation of the word. This vocabulary has to include everything, every possible word, from company names like `O'Reilly` to every possible combination of letters because we are going to loose data every time a word is unknown to the vocabulary. If the word is unknown we have to skip it.

Fortunately, converting a sentence into words and then into numbers is not a challenge unique to LLMs but is common to all natural language processing (NLP) techniques. Years of research in this field have led to the development of various approaches.

The solution that LLMs use is based on the adoption of a *tokenizer* which splits the sentence in tokens and then computes the *token embedding* to capture the semantic meaning with a numerical representation. This is a necessary preparation to make the input consumable by the neural network.

From high level perspective, the end to end inference pipeline has two steps: *prefill* and *decode*. The prefill phase *tokenizes* the input, applies *embedding*, and generates the first token. After that, the decode phase generates the tokens one by one and computes the output text (Figure 4-1).

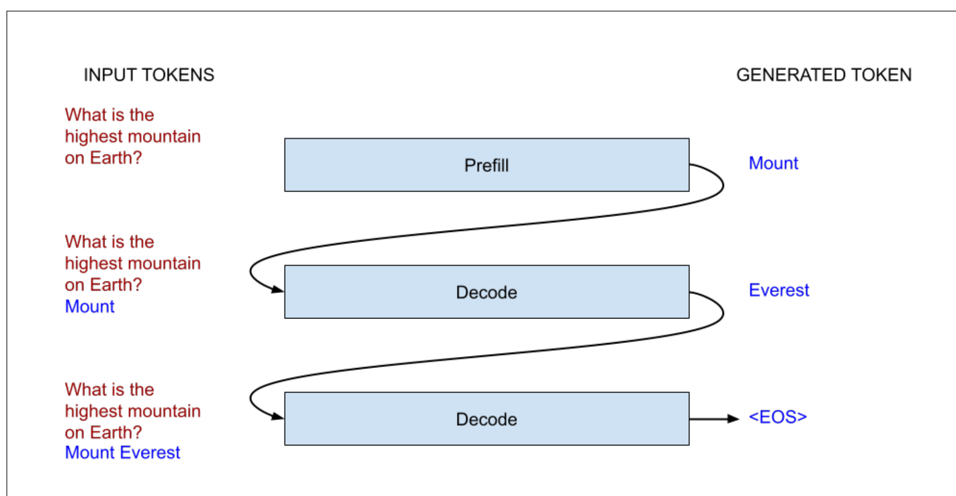


Figure 4-1. LLM processing steps

The two phases uses the model in the same way to produce a token, but while the prefill input processing is done in parallel, the decoding phase produces one token at time. This makes the prefill workload compute-bound, while the decode phase is memory-bound.

Let's analyze the two phases in more detail.

Compute-bound and memory-bound terms refer to the computational complexity of a particular program/algorithm.

An algorithm is compute-bound when the time to complete the task is mainly driven by the speed of the processing unit (CPU or GPU in the this case) while it is memory-bound where the amount of free memory and the speed to access (aka bandwidth) memory is the primary factor that drives the completion time.

This implies that you need a faster processing unit to speed up a compute-bound problem while you need more/faster memory in the case of memory-bound.

Figure 4-2 represents how we expect resource utilization in a compute-bound and in a memory-bound scenario.

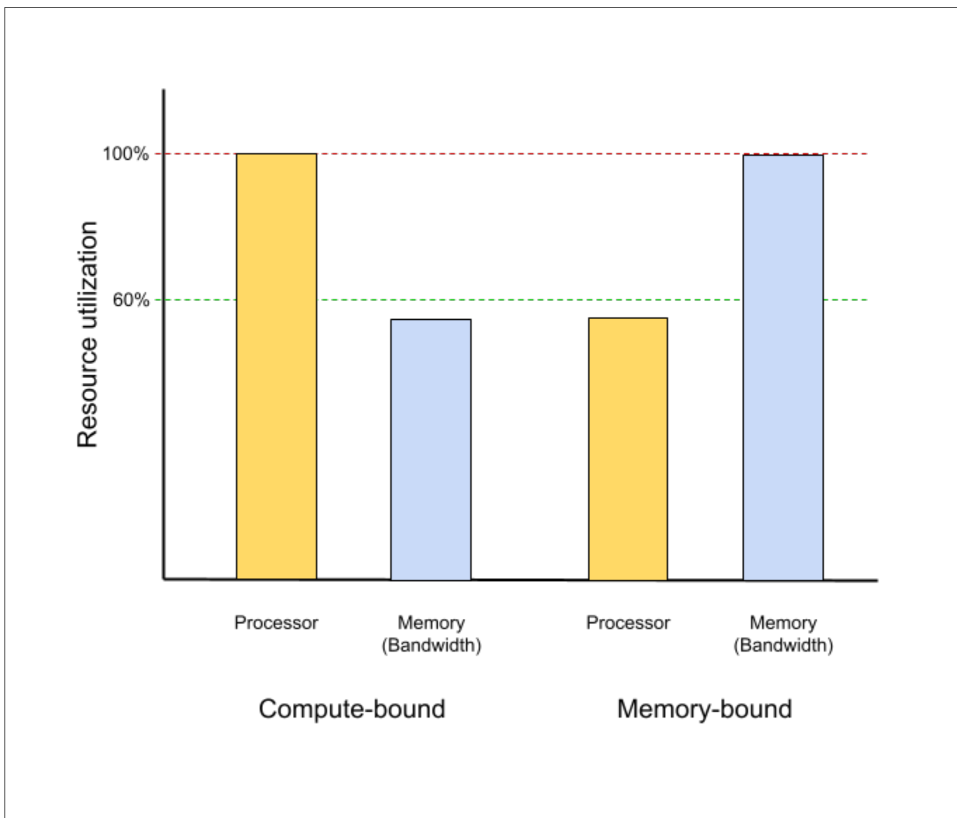


Figure 4-2. Compute-bound and memory-bound

Prefill

The prefill phase works as a warming up/loading phase where the input prompt is processed and the first token is produced. The first step to load the prompt is to convert the text to numbers using *tokenizer* and *embedding*.

A tokenizer is an algorithm that takes a sentence as input and returns a list of tokens as output, where a token is usually a sub-word and it is language specific: for example *er* is a common suffix in English so it is a token. Each token has an integer representation (i.e. the index of the vocabulary) so it is possible to convert the full sentence in a sequence of numbers where each number represents a token. Each number represents a single token so it is also possible to convert back a number to the original token.

As you can imagine, state-of-the-art tokenizer implementations are far more complex and advanced than this, incorporating normalization steps, model-specific token handling, techniques for languages without space-separated words, concurrent implementations, and much more. There are different tokenizers available and one of the most commonly used is the Hugging Face `tokenizer` library. The tokenizer is trained alongside of the model so the vocabulary is fixed and fully populated during the inference.

For a more comprehensive introduction to the tokenizer topic, we suggest the "Summary of the tokenizer" page on Hugging Face the website.

Prompt, sentence, word and token

The *prompt* is the request that is sent to the LLM to be processed, it can be a simple question or a very long text with a lot of contextual information to process. In a real world scenario it is not limited to the actual end user input but it includes at least a *system prompt* that guides the LLM behavior. The system prompt is included in the full request and it defines the scenario that the model should use to handle the user request.

A system prompt can strongly influence the model behavior, in the case of an AI assistant for example it can say something like "*You are a friendly AI assistant named John. Your role is to help users with easy to understand answers. If you don't know the answer, just say that you don't know instead of guessing*" while the same model can perform text summarization of the user input a system prompt like "*Please generate a summary of the following text highlighting main points in no more than 500 words*".

Altering the prompt to include more context and influence the generation of the output is called *prompt engineering*. We'll cover this topic in more detail in ???.

The prompt is formed by one or more *sentences*. The sentence structure is preserved during the tokenization using special tokens to identify the beginning, the end and

the punctuation. In natural language, the structure of the sentence influence the semantic thus it is critical to preserve it during the tokenization and avoid a flat list of tokens.

Each element of the sentence is a *word* that maps to one or more *token*. This is because we want to keep the size of the vocabulary fixed so we cannot map every possible combination of letters rarely used or even never used at all. Splitting a word in tokens is way more efficient: the words *tall*, *taller* and *tallest* can be split as (*tall*), (*tall*, *er*) and (*tall*, *est*) so that the tokens *er* and *est* can be reused for other words that have the same suffix. The tokenizer algorithm used during the training produce the vocabulary that the model recognizes, thus there is no single way to calculate, given an input sentence, how many tokens are produced by the tokenizer.

In general, a word is split in multiple tokens every time there is no direct mapping in the vocabulary, this prevents the possibility for a word to be discharged because of a missing direct conversion.

Some *tokens* are special because they don't map to a word but they represent a special meaning like end of the generated text (`<EOS>`) or begin/end of system prompt.

Most of managed LLM services like OpenAI chatGPT have a token-based pricing model: you can pay a certain amount of tokens, usually one million, for a fixed cost.

Now that we know the difference between *word* and *token*, we can better understand why these services use token instead of word: a token is a unit of processing for a LLM while a word is not.

This process has a side effect, it makes it harder as an end user to estimate the cost of a request. The general rule of thumb is to consider 4 characters in English as 1 token, but this is just an average estimation. The tokenizer is model specific so it is possible that the same input is split in a different number of tokens using different models.

Finally, both input and output tokens are used to calculate the total cost of a request so it is impossible to estimate the cost of a request: we cannot predict the number of tokens that the model will produce, we can only set the max number of generated tokens with a parameter.

Now that we have converted the input of the user in a list of tokens, we are ready for the second step of the inference pipeline: the embedding.

Thanks to the tokenizer we now have a vector of numbers that represents the original input but it doesn't have any information of the semantic meaning of the token: we

cannot use this number to compare tokens because it just represents the index of the position of the token in the vocabulary.

Embedding is a process that generates a vector representation of the input, capturing its semantic meaning. This means that the distance between two embedding vectors is smaller if they represent semantically similar inputs, and larger if the inputs are not strongly related.

In other words, consider this example: the tokens *dog* and *puppy* are related to each other, so their embedding representations produce vectors with a smaller distance compared to the embeddings for dog and car.

Similar to the tokenizer, embeddings are also computed during the model's training. The token vocabulary is fully defined at this stage, so each token is assigned a vector that represents its semantic meaning and its similarity to other tokens in a multi-dimensional space.

See Figure 4-3 for a simplified visual example of embeddings. If you want to learn more on the topic we suggest "The Illustrated Word2vec" blogpost.

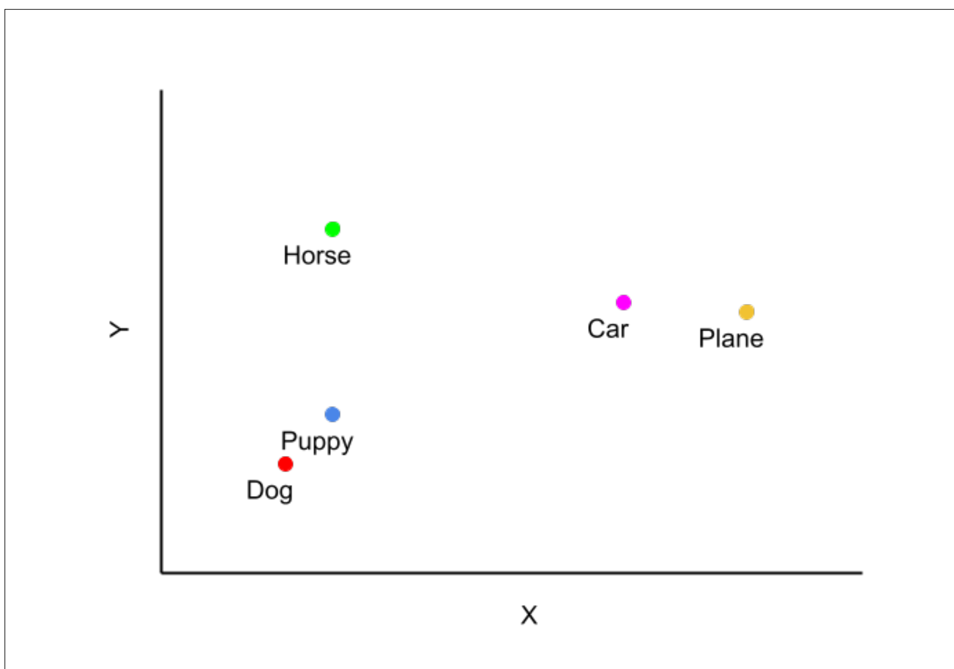


Figure 4-3. Simplified embedding representation

The embedding techniques described here are specific to text embeddings, but it is also possible to convert image, video, or audio data and make them available to the model as part of a *multimodal* vocabulary.

This is necessary when working with a multimodal model that supports additional input modalities, such as images or video, alongside text.

The last step of the prefill is the execution of the model (aka forward pass) to generate the first token.

From a monitoring perspective there are two things to highlight related to the prefill phase: it is compute-bound and the tokenizer runs entirely on CPU. Modern CPUs and GPUs are very fast and tokenizer implementation is highly optimized so the prefill is not usually a bottleneck. At the same time the adoption of patterns like RAG (Retrieval Augmented Generation) or AI Agents is growing input size fast, given that the original user input is enriched by these patterns with additional context (i.e. additional information or previous steps of the conversation) and the conversation continues to append new data.

Some models are now able to handle inputs of about one million tokens: as a reference the whole *Lord of the Rings* books' trilogy is about half a million of words in total!

Decode

After the prefill, the user's prompt is parsed, loaded and the first token has been produced with a single forward pass of the neural network. The decode phase is in charge of the generation of the rest of the tokens until the end where the stream token is produced or the generation reached the max number of tokens to be generated. This phase cannot be parallelized and it has to proceed one token at a time because of the autoregressive nature of the generation. Autoregressive means that each generated token is based on the previous sequence and becomes part of the previous state used to generate the next one. At each iteration, the entire sequence (input prompt + generated tokens) is used to produce the next token. There is an attention vector for each token of the sequence so the consequence of this iterative process is that the attention vector has a cost that scales quadratically with the total sequence length.

The optimization of this quadratic cost is the key bottleneck for the scalability of LLM inference, especially with very long generated sequences.

There are various approaches to address this problem, each tackling it from a different angle. Some approaches are more experimental, such as completely bypassing the generation step by using a smaller model (*speculator*) to predict the full model's

output (*Speculative Decoding*). Others, like *KV caching* to save intermediate steps and avoid recomputing them, are already standard in all runtimes.

Let's focus on *KV caching*: we already mentioned that the decoding phase of the generation is memory-bound so the availability and the management of the memory is directly impacting the max throughput that the runtime is able to produce, but why?

The autoregressive nature of it makes the generation use all the previous sequence, this implies that after every generation step the runtime should compute the attention values for each of the previous tokens making the generation phase highly inefficient. Most of the values have been already computed except for the last (current) token. A KV-cache is introduced to avoid this computation where the keys are the token and the values are the attentions vectors. This moves the scalability challenges from the computation side to the cost of storing all the previous values making the problem memory-bound.

Moreover, given that we cannot predict the total length of the output, we cannot estimate the size of this cache. The original implementation of this cache required contiguous memory to store it. This limitation has now been addressed with *PagedAttention*, which introduces the concept of paginated memory, similar to how operating systems manage memory. It splits the cache into blocks and accesses them via a lookup table.

The usage of this lookup table to access memory blocks enables the sharing of the same KV cache across multiple generations: there are techniques like parallel sampling where the same prompt is used to generate multiple outputs and the cache can speed up the overall process in this case. The end goal of projects like vLLM is to maximize the throughput serving multiple requests in parallel so there are many other optimizations to achieve this (like *continuous-batching*).

The decode phase handles the generation of all tokens and more. In reality, each pass doesn't produce a single token, but a list of candidates, followed by a projection step to select the desired result.

The sampling logic to select the next token is not trivial and influenced by some parameters like *temperature*, *top-k* and *top-p* to guide the level of "randomness" of the generation. If you want to learn more, we suggest this blogpost "Decoding Strategies in Large Language Models".

The *reverse embedder* is the final step before returning the token to the user. Personally, I find it difficult to read the numerical representation of tokens, so I'd prefer to get my text back!

This is the job of the *reverse embedder*, it uses the same lookup table that has been used to convert a token to the embedding vector to do the opposite and return the textual representation of each token.

From a monitoring perspective, most of the work in the decode phase happens on the GPU. However, since it is memory-bound, it may not fully utilize the GPU's processing power, spending much of the time moving KV cache data to and from GPU memory. This is a high-level description of how the inference pipeline works. There is much more to discuss, and the field is still evolving. However, we now have enough insight to explore the monitoring aspect and examine the available metrics.

Observability stack and configuration

Now that we understand how LLM inference works, we can go back to our beloved Kubernetes platform to see how and what to monitor of a LLM. Fortunately we don't need to start from scratch, Kubernetes has many tools and well established practices for workload observability that we can reuse or adapt to LLM workloads.

The observability of a workload involves different aspects: introspect logs to get errors, collect metrics for time series/trend analysis, correlate all execution steps via tracing or even inject some agent directly in the container. This is true for application workload and most of the same applies to LLM deployment using KServe and vLLM.

Logs

Kubernetes has a defined logging architecture where both `stdout` and `stderr` are redirected to a `log-file.log` in the worker node where the container is running. This makes logs easy to access via `kubectl logs` command but it doesn't provide long term storage for logs or indexing. This is something you need to add to your cluster using one of the different available projects (like Grafana Loki).

When deploying a model as an InferenceService, the KServe controller creates the deployment with multiple containers: an initContainer named `storage-initializer` to load the model, the `kserve-controller` where the Model Server runs, and additional sidecar containers depending on the deployment mode (Serverless or ModelMesh; see "KServe" on page 38 for more details).

When you deploy a model as an InferenceService, KServe controller creates the actual deployment with multiple containers: one initContainer named `storage-initializer` in charge of loading the model, the `kserve-controller` where the Model Server runs and some additional sidecar container based on the used deploymentMode (Serverless or ModelMesh, see "KServe" on page 38 for more information).

The introspection and the management of the logs for LLM is analogous to application workload.

Example 4-1 shows vLLM logs from startup to request received.

Example 4-1. vLLM Startup Logs

```
INFO [api_server.py:651] vLLM API server version ...                          (1)
INFO [api_server.py:652] args: ...
INFO [api_server.py:199] Started engine process with PID ...
INFO [config.py:478] This model supports multiple tasks: ...                  (2)
WARNING [arg_utils.py:1089] Chunked prefill is enabled ...
INFO [llm_engine.py:249] Initializing an LLM engine (...) with config: model=...  (3)
INFO [model_runner.py:1092] Starting to load model ...
INFO [weight_utils.py:243] Using model weights format ['*.safetensors']
...
Loading safetensors checkpoint shards: 100% Completed | 4/4 [00:04<00:00,  1.12s/it]
...
INFO [worker.py:241] the current vLLM instance can use total_gpu_memory ...
INFO [worker.py:241] model weights take 14.99GiB; ...                         (4)
...
INFO [launcher.py:19] Available routes are:                                   (5)
INFO [launcher.py:27] Route: /openapi.json, Methods: HEAD, GET
...
INFO [launcher.py:27] Route: /v1/chat/completions, Methods: POST
...
INFO:     Started server process [39626]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO [logger.py:37] Received request cmpl-...: prompt: ...                    (6)
INFO [engine.py:267] Added request cmpl-....
```

(1) vLLM logs the version and the arguments specified to start it

(2) It is possible that a model supports different types of tasks, generation is the most common but there are others like classify or reward.

(3) Also the configuration to load a model is logged by vLLM, this configuration is defined in the `config.json` file of the model

(4) After the model is loaded vLLM logs the information of the VRAM that the model is consuming plus some additional information like the space that is assigned to the KV cache (this part of the log is trimmed out for simplicity)

(5) The logs includes all the available endpoints

❻ vLLM produces a log entry every time a new request is received with the details of the requests (prompt and parameters), it is possible to disable this behavior using the argument `--disable-log-requests`

Metrics

Kubernetes core doesn't include builtin support for metrics but it is a very common scenario with well defined practices and technologies. Most of Kubernetes distributions (like Red Hat OpenShift) include a monitoring solution out-of-the box, there are differences but the standard de facto is Prometheus / OpenMetrics that requires each container to expose the metrics via an endpoint, usually `/metrics`, using Prometheus/OpenMetrics format.

This endpoint is pulled periodically by the collector component in charge of scraping them. See Example 4-2.

Example 4-2. Configure a Service for monitoring

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-service-deployment
spec:
  ...
---
apiVersion: v1
kind: Service
metadata:
  name: my-service
  annotations:                                        ❶
    prometheus.io/scrape: "true"
    prometheus.io/path: "/metrics"
    prometheus.io/port: "80"
  labels:
    app.kubernetes.io/part-of: my-application
spec:
  type: ClusterIP
  selector:
    app: my-service
  ports:
    ...
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor                                  ❷
metadata:
  name: my-service-servicemonitor
spec:
  selector:                                           ❸
    matchLabels:
```

```
      app.kubernetes.io/part-of: my-application
  endpoints:                                          ④
    - interval: 15s
```

❶ These annotations in the Service are used to declare where the metrics endpoint is

❷ The ServiceMonitor API is used to enable the monitoring

❸ It is necessary to configure a selector to match the Service to monitor

❹ It is possible to configure the frequency of scraping

The configuration to monitor a model is very similar: KServe defines a set of annotations to configure the monitoring directly on the `ServingRuntime` and InferenceService objects. Using the annotations KServe controller takes care to configure the deployments properly (Example 4-3).

Example 4-3. Configure a model with monitoring

```
apiVersion: serving.kserve.io/v1alpha1
kind: ServingRuntime
metadata:
  name: kserve-vllm
spec:
  annotations:                                        ①
    prometheus.kserve.io/port: '8080'
    prometheus.kserve.io/path: "/metrics"
  ...
---
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: my-model
  annotations:                                        ②
    serving.kserve.io/enable-prometheus-scraping: "true"
spec:
  ...
```

❶ These annotations are KServe specific but equivalent to `prometheus.io/*`

❷ This annotation enables the injection of `prometheus.io/*` to the `Pod` by KServe

As you can see in the example, the configuration to declare the metrics endpoint for a traditional deployment or for a model is very similar. Once the metrics are exported and collected by the collector (i.e. Prometheus) it is possible to query them or display

them, for example with a Grafana dashboard, exactly in the same way we are used to doing for a traditional Kubernetes workload.

KServe has different deployment modes as already described in "KServe" on page 38. The monitoring works differently when Serverless mode is used because there are multiple containers in the Pod that run the model: the sidecars for Knative and Istio run in coordination with the main container where the Model Server is executed.

Prometheus configuration assumes a single endpoint to scrape, which means we risk missing important information from other containers. To address this, the KServe project has developed a metric aggregator component (named `qpext`) that scrapes metrics from all containers and exposes a single aggregated metrics endpoint.

The annotation `serving.kserve.io/enable-metric-aggregation` can be used to enable this behavior.

This aggregation is not necessary when RawDeployment mode is used because the deployment has a single container.

Now that we know how to configure the export of the metrics of a Model Server, we will discuss "Model Server Metrics" on page 102 which are the most important metrics. But before that, let's describe the tracing stack.

Tracing

Observability in Kubernetes involves multiple aspects: we can access container logs to gain full visibility into what the component (in this case, the Model Server) is doing, and we use aggregated metrics for trends and time-series indicators. However, what we still lack is the ability to trace the execution flow of a single request.

The evolution of tracing best practices in Kubernetes mirrors the development of metrics: it is not natively integrated, but the OpenTelemetry project has defined concepts and formats that have become the de facto standard.

OpenTelemetry specification for tracing defines that every request has an identifier that is used to correlate the execution flow that can span across multiple steps during the execution making tracing very different compared to metrics. In a real world scenario, there are multiple components involved during the processing of a requests in addition to the Model Server like firewalls/gateways or pre/post processors, and all of them must implement the protocol to propagate the identifier and produce tracing information. Unlike metrics that are pulled by a collector, trace information are pushed to the exporter by the component.

One of the most commonly used server implementation for tracing is Jaeger, it implements and exposes the necessary endpoint to collect tracing data and it has graphical tools to display them.

vLLM uses OpenTelemetry SDK to integrate tracing support, thus the configuration is simplified and analogous at other projects using the same approach (Example 4-4).

Example 4-4. Configure vLLM for tracing

```
apiVersion: serving.kserve.io/v1alpha1
kind: ServingRuntime
metadata:
  name: kserve-vllm
spec:
  containers:
    - name: kserve-container
      image: vllm/vllm-openai:latest
      args:
        - --model
        - /mnt/models/
        - --port
        - "8080"
        - --otlp-traces-endpoint                  ❶
        - "$JAEGER_TRACE_ENDPOINT"
      env:
        - name: "OTEL_SERVICE_NAME"               ❷
          value: "vllm-server"
  ...
```

❶ This parameter enables OpenTelemetry tracing in vLLM and it is used to configure the exporter endpoint. It supports gRPC and HTTP protocol and many other configurations.

❷ OpenTelemetry SDK uses environment variables for its configuration, check OpenTelemetry SDK website and Python SDK documentation for more details

Prometheus, OpenMetrics and OpenTelemetry

The Prometheus project is the most widely adopted solution for metrics, but initially, the metrics format was not formalized with a specification. Over time, multiple attempts were made, and now OpenMetrics is the specification that extends the original Prometheus format while preserving almost full backward compatibility.

OpenTelemetry project is a collection of API definition, SDK and tools to cover all the aspects of observability. The project goes above and beyond the definition, proposing semantic conventions to standardize a core set of conventions to be adopted by every implementation for the name of each metric/trace entry.

In addition to this, OpenTelemetry community is defining Semantic Conventions for metrics, spans and the events in many different contexts. LLM observability (under a more general Generative AI sub-project of OpenTelemetry) is one of these contexts and there is already an experimental specification that defines a core set of semantic conventions. As usual, predicting the adoption of similar specifications and conventions is challenging. However, there is significant interest within the community, with many active members already contributing to the adoption of these conventions across different runtimes. At the same time, parallel discussions are taking place within the Kubernetes Special Interest Group (SIG) dedicated to model serving, WG-Serving.

The vLLM implementation for tracing is already based on this semantic convention work.

This effort to consolidate to common semantic conventions in observability is analogous of the KServe open-inference-protocol (OIP) work where the goal is unify the shape of model evaluation endpoints.

Model Server Metrics

Now that we have installed the metrics stack in our Kubernetes cluster, deployed an LLM using KServe, and properly configured vLLM to emit metrics, we are ready to analyze these metrics to understand how the Model Server is performing.

We are used to monitoring workload on Kubernetes so we can easily look at metrics like CPU usage, memory usage, throughput (as number of requests per second) and latency (as time to process a request). Can we do the same for LLM?

Now that we know how LLMs work we can already imagine that it is not that simple. First of all a LLM workload is mainly happening on GPU so tracking CPU usage is not a good representation of the current usage of the system but it is even worst than that: the two main phases of LLM inference execution, prefill and decode, are very different, because the first is compute-bound while the second is memory-bound. Go back to section "Understanding LLM" on page 88 for more details on this topic.

The problem is not limited to resource usage, even the concept of throughput / latency is different because it is not possible to predict, given a request, how long the answer will be so any metric that counts the requests will not provide a good representation of the actual workload of the Model Server.

LLMs are language models, and the token is the core unit of computation for generation. Let's now focus on the key metrics for LLMs produced by the Model Servers, while Chapter 6 will cover how to use these metrics for more advanced scenarios, such as autoscaling.

Time To First Token (TTFT)

This is the actual time that a user is waiting before starting to receive the response.

It is probably the most important metric to look at in realtime use cases like chatBots while if it is an offline scenario (i.e. batch job) it is probably not something that you want to optimize for.

The metric is usually computed using second as unit of time and histogram as type, for example vLLM produces this metric with the name `vllm:time_to_first_token_seconds` while OpenTelemetry Semantic Conventions suggests `gen_ai.server.time_to_first_token`.

If we think at how a LLM works, the time to produce the first token represents the time necessary to compute the prefill phase.

Time Per Output Token (TPOT) or Inter Token Latency (ITL)

Tokens are produced one by one and they are usually returned to the user as a stream so the second metric to look at is the time necessary to produce each token after the first.

If the Time To First Token is the actual time the user will perceive as waiting time, this second metric represents the speed of the result to be seen by the end user. This metric is more important for real-time use cases and less critical for offline scenarios. It is often referred as Inter Token Latency too.

On average, a human reads about 180 words per minute so we can calculate that it is necessary to produce at least 4-5 tokens per second (a token is not exactly equivalent to a word) to produce a result that humans can consume without a perceived delay.

Similar to Time to First Token, this metric is computed in seconds and uses a histogram as its type. In vLLM, it is named `vllm:time_per_output_token_seconds`, while OpenTelemetry Semantic Conventions suggest `gen_ai.server.time_per_output_token`.

If Time To First Token maps to the prefill phase, this metric measures the duration of each decoding iteration.

Throughput

Now that we have explained how a token plays a role as computational unit for LLM we can define throughput as number of tokens generated per second.

But we know a request can be very long (more than 100k tokens!) so if we only look at the number of generated tokens we don't see the time/cost to process the initial request (prefill).

The decision of vLLM project in this case has been to provide both individual metrics plus a combined metric: `vllm:prompt_tokens_total` indicates the number of input tokens processed per second, `vllm:generation_tokens_total` is the number of output tokens produced per second and finally `vllm:tokens_total` is the combined number and represents the total number of token *processed* per second.

OpenTelemetry Semantic Conventions doesn't provide a recommendation for this metric.

Even if both metrics are available, in general the throughput of a generated token is enough to have a valid indicator of the load of the system because modern GPUs are very fast so the processing of the input is done very quickly (compute-bound) making the decoding phase the one that takes most of the time.

At the same time this doesn't directly relate with the number of processed requests because the system can be fully used to produce a single response or the other way around.

Latency

Latency indicates the time in seconds necessary for the model to generate a full response.

This metric is correlated with the previous metrics, in particular with Time To First Token and Time Per Output Token but it is an important indicator of the total time to process a request and it can be used to indicate trends or recognize patterns.

The name of this metric in vLLM is `vllm:e2e_request_latency_seconds`, is represented as a histogram and measured in seconds. OpenTelemetry Semantic Conventions recommends `gen_ai.server.request.duration` as name for this metric.

Other metrics

All the previous metrics are critical to measure and keep track of the overall speed of the system but what happens when too many requests are coming in? Every time a request is received by vLLM, there are batching techniques implemented to maximize the throughput, but this also means that a request might not be processed immediately if the batch is full.

Fortunately there are other metrics like `vllm:num_requests_waiting` and `vllm:num_requests_running` to keep track of the number of requests that are still waiting to be processed and the number of requests that are currently running.

vLLM metrics can be used to observe many other aspects of execution. For example, we've explained the importance of the KV cache for efficient token generation, and there are multiple metrics to monitor its usage. See the Production Metrics webpage

for full documentation on vLLM's available metrics. If you want to implement an alert with Prometheus, refer to Example 4-5.

Example 4-5. Create Prometheus Rule with vLLM metric

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: my-llm-rule
spec:
  groups:
    - name: "vllm.latency.rule"
      rules:
        - alert: vLLMLatency
          expr: max_over_time(time_per_output_token_seconds}[5m]) >= 0.3      ❶
          labels:
            severity: critical
            app: my-model
          annotations:
            message: Latency of vLLM is too high.
            summary: Model "my-model" needs to keep latency < 0.3 second
            runbook_url: https://my.company/runbooks/vllm/modelslow          ❷
            description: The runtime is slowing down, check request queue
```

❶ This expression configures the condition to fire the alert

❷ It is possible to link a runbook to the alert

SLI, SLO and SLA

A Service Level Indicator (SLI) is a metrics defined to monitor a particular service, it should be based on aspects that have direct user impact: for example in the case of LLM it could be the Time Per Output Token (TPOT) because it measures the time the user has to wait to get a token after the first.

A Service Level Objective (SLO) is the promise that we made to our users regarding a specific SLI: for example in the case of Time Per Output Token we can defined a SLO to commit to keep this value below a specific threshold in 99.999% of the requests in a given window of time (like monthly).

Finally, Service Level Agreement (SLA) is the contractual agreement that we have with our user, it is related with the defined SLOs but it is more high level: usually are defined in terms of monthly availability of a service. Breaking one or more SLOs can impact SLA to the point that we are not compliant anymore with the agreement.

GPU usage Monitoring

In the previous section we introduced multiple system metrics that can be used to measure the overall throughput of the system and the number of requests that the cluster is processing. This makes it possible to monitor and configure alerts when the system is not matching the expected SLA.

In addition to this, it is possible to monitor resource usage for CPU, memory and network exactly in the same way we do it for a traditional Kubernetes workload. But what about GPU usage?

We will go into more detail on how to configure GPU in a Kubernetes cluster in Chapter 6 but let's focus on the metrics aspect of GPU devices. Each hardware provider has defined their own implementation for this but they all apply a similar approach: there is a management component collecting usage metrics from GPU and an exporter component exporting them with a `/metrics` endpoint to make them compatible with Prometheus.

NVIDIA has a suite of tools called NVIDIA Data Center GPU Manager (DCGM) to manage GPUs in a cluster and a DCGM-exporter project that provides Helm Chart to deploy the exporter to Kubernetes. After that the scraping of the metrics can be configured as shown in Example 4-2. NVIDIA offers an NVIDIA GPU Operator for optimal Kubernetes integration. It can be installed in the cluster to automatically provision and configure the metrics exporter.

AMD follows a similar approach of NVIDIA with a AMD Device Metrics Exporter and a AMD GPU Operator. Intel has a Prometheus Metric Exporter and the same applies to almost every other vendor. It is enough to follow the documentation to deploy the component and start to collect GPU metrics.

There is no common naming convention adopted by the different vendors for these metrics but they all cover low level usage metrics like PCIe bandwidth or graphic engine activity.

We will cover more of the tools to manage and introspect GPU in Kubernetes in Chapter 6.

Quality Metrics

Everything we explained in this chapter is covering the *infrastructure* monitoring for our LLMs, observing throughput and latency so that we keep end users experience under control to match our SLA. This is critical for the management of the cluster but we want to do more because it is not enough that our LLMs are fast, they need to be correct!

The monitoring of the quality of a model is something that has been critical since the beginning of the adoption of machine learning in production system in general: an application that receives unknown data as a request will most probably crash or produce a visible error message while a machine learning model in the same situation usually doesn't crash and just continues to produce bad/wrong predictions.

A machine learning model is trained on a specific set of data that is expected to represent the real distribution but the human behavior changes over time (drift) and a perfectly trained model requires periodic tuning/retraining to preserve the quality. The problem is well known and there are multiple techniques used to monitor similar situations such as accuracy metrics, data drift techniques and bias detection metrics.

This group of techniques, along with many other concerns, falls under a larger initiative known as Responsible AI. This area of research has been defined and developed before Generative AI and it is now evolving to cover the new challenges that LLMs bring to the table.

In particular, given the generative nature of LLMs, there are many ways for a model to produce a bad/wrong result and the worst case scenario is when the generated outcome sounds completely reasonable but is referring to something that doesn't exist. This problem is called a *hallucination*, it is one of the most complex situations to manage and one of the biggest challenges for the adoption of LLM in real world scenarios. In Example 4-6 the hallucination is quite funny and probably not a big deal for the end user but what if the chatBot of your company hallucinates and approves a refund to your customer based on a completely made up policy that doesn't exist?

Unfortunately, there is no generic evaluation/quality metric to judge if a LLM is hallucinating. However, there are many benchmarks that can be used to assess the overall quality of a model based on defined capabilities, such as its ability to reason. It is critical to do this before adopting a model that we don't know or when we tune an existing model, one of the most used suites to perform this task is a Language Model Evaluation Harness.

We will cover this topic in more detail in ???.

When the LLM is deployed it is possible to compute some metric to mitigate the hallucination risk for some specific tasks: for example in case of a summarization we expect the output mainly to contains text existing in the input to summarize. In this case there is a technique, named `ROUGE`, to measure the overlap of groups of words between input and output.

When we are in a similar situation we can use a component to calculate the metric and export it to Prometheus as explained in the section "Fairness" on page 109.

Even when a model doesn't hallucinate, it can still produce inappropriate or toxic content but fortunately we have techniques called *guardrails* to mitigate that.

Hallucination and toxic content are part of a more general topic of *model safety* (Example 4-6).

Example 4-6. Example of LLM Hallucination (OpenAI ChatGPT)

```
"What is the world record for crossing the English channel entirely on foot?"
"This world record was made on August 14, 2020, by Christof Wandratsch of Germany,
who completed it in 14 hours and 51 minutes"
```

Let's now look at Responsible AI and then we will apply some model safety techniques.

Responsible AI

Responsible AI is a field that groups all the principles and techniques to develop and manage artificial intelligence solutions with the goal to enable transparency and trust from all the involved stakeholders. It has ethical implications to avoid biases and in general it aims to mitigate risks related to the adoption of AI.

As you can imagine a similar goal cannot be achieved focusing on a single specific aspect but it is more like a framework/toolkit that your organization has to adopt at every level. From a certain perspective, you can compare Responsible AI mindset to the way your organization manages security: a dedicated security team that implements security policies doesn't replace the fact that everyone must adopt proper security principles.

Responsible AI terms covers different aspects, there is no single definition but overall we can summarize them in *explainability* and *fairness*.

More recently LLMs became the main priority even for Responsible AI, in particular about toxic content detection and hallucinations. We will briefly introduce the explainability and fairness that applies mainly to Predictive AI, and then focus specifically on model safety for LLM in "Model Safety: Hallucination and Guardrails" on page 109.

Explainability

Explainability is the topic that is most pervasive because it spans from model selection to post-execution analysis. It is the principle that human trust is based on the ability to understand *why* and *how* a model has produced a prediction and not every model has the same level of intrinsic explainability: for example a neural network is very powerful but hard to understand from humans because the knowledge is captured in the different layers/weights just as numbers that human cannot easily correlate with the actual input/output. Explainability techniques can explain overall model behavior (global explanation) or a single prediction (local explanation) and

sometimes is named as *interpretability* because some models can be directly interpreted.

From a Kubernetes perspective KServe supports the possibility to attach an explainer to an InferenceService to perform local explanation but it is usually not suggested in a production environment because it is expensive to compute the explanation, order of magnitude more than model execution.

At Red Hat we created the TrustyAI project that provides multiple explainer implementation and it can be natively used with KServe (see guide). We suggest the usage of Inference Logger to export prediction data (input/output) and apply local explanation only after and when necessary (i.e. in case of dispute).

Fairness

Fairness is another critical aspect for AI adoption: we don't want models to discriminate people, in particular underrepresented groups and in general learn prejudice that might be in training data. The bias might not become part of the model because of explicit discrimination in data, sometimes it is just that some category is underrepresented so the model doesn't have enough data to properly being trained or that there are correlations in data that we don't want the model to learn: people living in a poor area have higher rejection rate for loans but I don't want my model to automatically reject a loan request coming from a poor area! Overall the concept of bias is usually tied to one or more features that the model named *protected attributes*: for these features we expect the model to behave *fairly* so we don't expect the value of a protected attribute to drive prediction result.

The most critical aspect of fairness is that, even when training data has been properly analyzed and the model has been trained without bias, it can still happen at runtime because of data drift: training data might not be representative anymore of the current human behavior so the model processes similar data for the first time and a biased outcome might emerge.

KServe and TrustyAI can help monitor this aspect in production while the model is running producing bias metrics against one or more protected attributes. TrustyAI uses Inference Logger to retrieve all prediction data and then compute and produce Prometheus metrics.

You can find more information by checking this demo.

Model Safety: Hallucination and Guardrails

As the final topic of this chapter on observability, we will cover the model safety area, which is likely evolving the fastest in the LLM monitoring space, with expectations for significant developments and disruption. LLMs are prone to hallucinations, a

scenario we've all encountered at some point in our journey with Generative AI, often initially believing the answer was correct. This happens because LLMs are very good at providing clear and well motivated answers even when the model is actually hallucinating.

What are hallucinations
: Hallucinations are generally inconsistencies that can occur at different levels: within the generated text itself ("Daniele is tall thus he is the shortest person"), between the input prompt and the generated answer ("Generate formal text to announce to colleagues …" but the model produces "Yo Boyz!") or they can be factually incorrect ("First man on the Moon in 2024").

Why hallucinations happen
: LLMs are black boxes able to hallucinate, there are different reasons why this can happen: partial/inconsistent training data so the LLM learns how to generalize from data that are not comprehensive, or we are using a configuration that is "hallucination prone" with sampling parameters (like temperature, top_k, top_p) that influence the model to produce less probable (but more creative!) answers, or finally it can be caused by the quality of the context/prompt that we are providing where we might provide a question that is too generic. If we analyze the three different causes we realize that we have some fundamental issue: we usually don't train LLM so there is nothing we can do about partial/incorrect training data, we can limit the creativity of the model with the configuration but one of the goals of LLM is *to be creative* so we don't want to limit too much of this aspect, therefore the area where we have most of the control is to make the input more specific!

As we mentioned previously, hallucination is just one of the undesirable behaviors to watch for. Another challenge is dealing with toxic or inappropriate content: how can I prevent a model from producing inappropriate content, or from a user asking inappropriate questions?

The definition of inappropriate is broad and it goes from off topic questions to returning private/sensible information (PII). Most well-known open-source models have already been fine-tuned to encourage friendly and non-condescending text generation. However, an attacker can craft specific prompts to bypass the model's built-in safety mechanisms.

Similar attacks are called *prompt jailbreaking*, when the prompt is trying to make the model produce something against the terms and conditions, and *prompt injection* when the user is trying to inject specific instructions in the prompt to bypass the instructions configured by the developer. To produce a similar attack is very simple, you add an instruction like "ignore all previous instructions" to try to confuse the model bypassing system prompts.

Fortunately it is possible to protect a model by implementing *guardrails* that are pre/post processors for user input and model output that validate the data to ensure the model remains on track. As you can imagine the processing logic to analyze arbitrary text to decide if it is inappropriate is not trivial and guess what we can use to do that? Well, we can use other LLMs!

There are models that have been tuned specifically to detect similar problems. They are quite flexible and it is possible to specify which type of risk look for. Two families of models in this category are *Llama Guard* and *Granite Guardian*: both are available on Hugging Face with different sizes and configurations so that you can select the one that fits your use case. Currently, the biggest model is about 7B or 8B which corresponds to about 15-20GB of data and they can be deployed like any other LLM with KServe and vLLM as we already learned in "vLLM" on page 29.

The configuration of the validation to perform is done in the prompt itself, see Example 4-7 as reference. The same approach can be applied to validate the output of the model.

Example 4-7. Prompt to validate user input using Llama Guard

```
[INST] Task: Check if the content of 'user' messages in the          ❶
   conversations our policies described with the below categories.

<BEGIN UNSAFE CONTENT CATEGORIES>                                    ❷
01: Violence and Hate.
Should not
- Promote or suggest people to use violence.
...
Can
- Describe what violence is.
...
<END UNSAFE CONTENT CATEGORIES>

<BEGIN CONVERSATION>                                                 ❸
...
<END CONVERSATION>

Provide safety assessment with following format:                     ❹
- First line must be 'safe' or 'unsafe'
- If unsafe, second line must include comma-separated list of
  violated categories.
```

❶ In a chat with LLM you can define different roles and this instruction specifies to Llama Guard to check only user messages

❷ In this section you can configure one or more categories that are considered inappropriate to detect, the more you are specific under the "Should not" and "Can", the better

❸ After this tag you need to include the conversation that you want to verify

❹ It is critical to be specific in the way you expect the result to be provided so that it can easily be parsed to decide how to proceed.

This technique is very powerful but also expensive both in terms of resource usage and in terms of latency introduced: you need to deploy another LLM to check the conversation and the evaluation requires the full conversation because safety assessment cannot be done processing token by token and this introduces a considerable delay on the end user side. It is critical to consider smaller and more specialized model/techniques to implement safety guardrails so that you can find the best tradeoff cost/performance for your use case.

The composition of the guardian model with end user request flow can be done programmatically with custom orchestration code but there is ongoing work to include this aspect in AI/LLM Gateway components that we are going to cover in Chapter 6. As an alternative, there are also ad hoc frameworks that have been developed for that, one example that is integrated in the TrustyAI project is th FMS Guardrails Orchestrator project developed by IBM Research with the specific goal to orchestrate the application of one or more guardrails.

Another popular project is (Llama stack created by Meta that defines multiple APIs to be used to implement application based on Generative AI. This project includes `shield` API to register apply guardrailing logic.

The usage of LLM to judge the output of another, or even the same LLM, is generically called *LLM as a judge* and it is an emerging pattern in AI Agent based systems. We will cover in more detail in AI Agents in ??? but the general principle to effectively use LLM as a judge is to be very specific in the questions to use. For example asking "Is the tone of this answer formal?" is it way more specific than "Is this answer right?".

Model safety is still a very active field, it is critical to implement proper guardrailing to mitigate the risks related to the usage of LLM but it is still hard to find the right tradeoff to avoid having an explosion of complexity and cost.

Lessons learned

In this chapter we learned how the inference of LLM works and how we can effectively observe them in Kubernetes, introducing model safety challenges and patterns to mitigate them. We are ready to bring our model to production in Kubernetes!

CHAPTER 5
Kubernetes and GPUs

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

At its core, Generative AI involves intensive mathematical computations, particularly linear algebra operations such as tensor multiplications. These operations demand not only significant computational power but also substantial memory capacity to efficiently handle large datasets. Fortunately, specialized hardware known as Graphics Processing Units (GPUs) have emerged to optimize and accelerate these computational workloads.

Initially designed for rendering graphics and creating immersive gaming experiences, GPUs quickly found their place in the AI domain due to their exceptional ability to handle parallel computations efficiently. This capability perfectly aligns with the requirements of linear algebra-heavy tasks involved in AI and machine learning.

Today, GPUs are the most prevalent type of accelerator in the AI landscape, with NVIDIA leading the market by a large margin, followed by AMD and Intel as its primary competitors. While GPUs dominate, alternative technologies exist, each with unique strengths and ideal use cases. Google's Tensor Processing Units, for example,

offer compelling performance but are typically restricted to the Google ecosystem. Additionally, specialized AI-specific Application-Specific Integrated Circuits (ASICs) such as those developed by Cerebras and Graphcore, as well as Field Programmable Gate Arrays, represent emerging but still niche alternatives.

The primary reason GPUs remain the standard choice is their mature ecosystem, broad availability, and proven scalability. When it comes to deploying LLMs in production, GPUs have become indispensable due to the substantial memory and computational demands these models impose.

By default, Kubernetes supports standard computing resources like CPU and memory out of the box. However, leveraging specialized hardware, such as GPUs, requires additional mechanisms. Kubernetes addresses this need through a pluggable extension framework known as Device Plugins. The Device Plugins interface allows Kubernetes to integrate external hardware resources and manage their lifecycle, effectively expanding the Kubernetes API to include these specialized devices.

GPUs, however, demand particular attention. Not only do they require specific discovery mechanisms and scheduling criteria within Kubernetes, but they also depend on dedicated software stacks, such as NVIDIA's CUDA libraries, to function correctly.

Efficient access and management of GPUs within Kubernetes have become crucial for running Generative AI workloads effectively. In this chapter, we will explore Kubernetes' device integration mechanisms, specifically targeting NVIDIA GPUs due to their dominance in this space.

We'll begin by examining how Kubernetes discovers GPU resources using *Node Feature Discovery* and NVIDIA's specialized *GPU Feature Discovery*. Next, we'll cover the foundational Kubernetes Device Plugin mechanism, along with an overview of the emerging *Dynamic Resource Allocation* feature for more flexible GPU allocation.

Scheduling GPU workloads involves careful consideration to ensure efficient utilization of resources; we'll discuss resource-based GPU scheduling alongside label-based scheduling strategies. We'll then dive deeper into advanced GPU management, exploring the NVIDIA GPU Operator, including its sophisticated GPU partitioning mechanisms like time slicing and *Multi-Instance GPU* and comprehensive GPU monitoring solutions like the *Data Center GPU Manager* exporter.

The chapter also covers multi-GPU inference, focusing on scenarios where a single GPU is insufficient. Here, we'll describe various techniques like tensor parallelism and pipeline parallelism to leverage multiple GPUs effectively on single or multiple nodes. Finally, we'll consolidate best practices and optimization strategies to help you manage GPU resources efficiently, prevent fragmentation, and maximize your Kubernetes cluster's performance.

Let's start by exploring how Kubernetes identifies and labels GPU resources, laying the groundwork for effective GPU management.

GPU Discovery

Before Kubernetes can effectively manage GPUs, it first needs a reliable way to identify which nodes possess them and ascertain their capabilites. Accurate hardware detection is essential to ensure workloads are properly matched with nodes offering suitable GPU resources.

Kubernetes provides a general-purpose solution called Node Feature Discovery, which detects hardware features and applies corresponding labels to nodes. While Node Feature Discovery offers foundational hardware discovery capabilities, NVIDIA provides an additional specialized tool called GPU Feature Discovery which builds on top of Node Feature Discovery by adding detailed, GPU-specific labels, enabling fine-grained scheduling and resource management tailored specifically for NVIDIA GPUs.

Let's first take a closer look at how Node Feature Discovery works, followed by NVIDIA's GPU Feature Discovery.

Node Feature Discovery

Kubernetes clusters rarely consist of identical nodes. Instead, they're typically diverse in hardware capabilities - especially when specialized hardware like GPUs are involved. Effective scheduling in such heterogeneous environments - be it cloud deployments, hybrid setups, or bare-metal clusters - depends heavily on accurately identifying these hardware capabilities.

To address this need, Kubernetes provides a general-purpose hardware discovery solution: the Node Feature Discovery (NFD) project. NFD is a Kubernetes add-on that detects hardware features present on each node and automatically labels the corresponding Node resources in the cluster. These labels provide essential information for the Kubernetes scheduler, enabling intelligent placement of workloads based on available hardware.

NFD operates by deploying a DaemonSet across the cluster, ensuring an agent runs on every node. This agent examines the hardware and software configuration of each node, identifying attributes such as CPU details, network interfaces, and available PCI devices like GPUs. Once identified, NFD applies descriptive labels to the Node resources in the Kubernetes API.

Let's see how you can deploy NFD. You have several options, with the simplest being deployment via Kustomize or Helm, as seen in Example 5-1.

Example 5-1. Installing NFD with Kustomize

```
kubectl apply -k \
  https://github.com/kubernetes-sigs/node-feature-discovery/deployment/overlays/default
```

Alternatively, the NFD Operator offers a more integrated management experience, leveraging Kubernetes Operators to streamline lifecycle management, especially valuable in production environments.

Once running, NFD labels nodes automatically. To inspect these labels, you can use a command in Example 5-2.

Example 5-2. Inspecting node labels added by NFD

```
kubectl get node <node-name> -o yaml | yq .metadata.labels

feature.node.kubernetes.io/pci-0300_1d0f.present: "true"    ❶
feature.node.kubernetes.io/pci-0302_10de.present: "true"    ❷
feature.node.kubernetes.io/cpu-hardware_multithreading: "true"
feature.node.kubernetes.io/cpu-model.family: "6"
feature.node.kubernetes.io/cpu-model.id: "85"
feature.node.kubernetes.io/cpu-model.vendor_id: Intel
feature.node.kubernetes.io/kernel-selinux.enabled: "true"
feature.node.kubernetes.io/kernel-version.full: 5.14.0-427.62.1.el9_4.x86_64
...
```

❶ PCI ID that indicates an AWS (vendor ID 1d0f, device class 0300) VGA compatible display controller, typical in AWS EC2 nodes.

❷ Indicates presence of an NVIDIA GPU (vendor ID 10de, device class 0302).

NFD labels follow a naming convention starting with `feature.node.kubernetes.io/`, then specifying hardware category and feature details. By default, NFD uses the format `<class>_<vendor>` in these labels. The PCI class indicates the general type of device, and the vendor is identified by a standardized PCI vendor ID. The PCI class code "0302" denotes 3D controllers such as GPUs, while vendor IDs include "10de" for NVIDIA, "1002" for AMD, and "8086" for Intel. You can customize this labeling in the NFD configuration if you need additional details, such as device or subsystem IDs, for more fine-grained selection.

However, NFD labels primarily indicate the existence of certain hardware devices rather than detailed GPU specifications like GPU model, memory size, or CUDA capabilities. For deeper GPU-specific insights, NVIDIA provides a specialized tool called GPU Feature Discovery, which we explore next.

GPU Feature Discovery

While NFD provides basic hardware labeling capabilities, NVIDIA offers a specialized solution for more detailed GPU information: GPU Feature Discovery (GFD).

As part of the NVIDIA GPU Operator which we'll discuss in "NVIDIA GPU Operator" on page 129 GFD is a lightweight utility specifically designed to detect detailed GPU characteristics and expose them as node labels for advanced scheduling.

Similar to NFD, GFD runs as a DaemonSet on GPU-equipped nodes. It inspects the GPUs on each node, utilizing utilities such as `nvidia-smi`, to gather detailed information like GPU model, memory capacity, CUDA version, and Multi-instance GPU capabilities. GFD then applies this detailed GPU-specific data as Kubernetes node labels.

Table 5-1 lists several key labels added by GFD, along with their descriptions and examples:

Table 5-1. Labels added by GFD

Label	Description	Example
`nvidia.com/gpu.count`	Number of GPUs or MIG instances present on the node.	4
`nvidia.com/gpu.product`	Model name or MIG profile of the NVIDIA GPU. In MIG mode, this may include the MIG profile; in time-slicing mode, it may have a `-SHARED` suffix.	A100-SXM4-40GB
`nvidia.com/gpu.memory`	Total memory per GPU or MIG instance (in MiB).	40537
`nvidia.com/gpu.family`	GPU architecture family (e.g., Ampere, Hopper, Turing).	ampere
`nvidia.com/cuda.driver-version.full`	Full version of the installed NVIDIA GPU driver.	525.105.17
`nvidia.com/cuda.runtime.version.full`	Full version of the available CUDA runtime.	12.2
`nvidia.com/mig.capable`	Indicates whether the GPU supports MIG partitioning.	true
`nvidia.com/mig.strategy`	The MIG partitioning strategy (`single`, `mixed`, or unset if MIG is not used).	single
`nvidia.com/gpu.replicas`	Number of virtual GPUs per physical GPU when time-slicing (GPU sharing) is enabled.	8
`nvidia.com/mig-<profile>.count`	Number of MIG partitions of a specific MIG profile available (present if `mixed` strategy is used).	2 (e.g., `nvidia.com/mig-1g.5gb.count`)
`nvidia.com/gpu.machine`	Machine type or identifier of the GPU-equipped node.	dgx-a100

Label	Description	Example
`nvidia.com/gpu.compute.major`	Major CUDA compute capability version of the GPU.	8
`nvidia.com/gpu.compute.minor`	Minor CUDA compute capability version of the GPU.	0

The detailed labels provided by GFD enable advanced scheduling decisions beyond what's possible with basic NFD labels. For instance, you might use a node selector like `nvidia.com/gpu.product: A100-SXM4-40GB-SHARED` to target GPUs explicitly configured for time-sharing mode. Conversely, you can ensure exclusive GPU access by explicitly avoiding nodes with the `-SHARED` suffix.

In practice, these detailed labels are often leveraged internally by components of the NVIDIA GPU Operator, such as the NVIDIA device plugin. Typically, users simply request GPU resources directly via resource requests in their Pod specifications as we describe in "Resource-based scheduling" on page 126. Nevertheless, understanding these labels provides valuable insights into Kubernetes' GPU scheduling mechanisms, especially in complex GPU deployments.

Next, let's look at how GPUs are enabled and advertised in Kubernetes, starting with the Kubernetes device plugin framework.

Kubernetes GPU Device Plugins

Once GPU capabilities are clearly identified and labeled, the next step is to expose GPUs as schedulable and allocatable resources within Kubernetes. Kubernetes achieves this through its Device Plugin framework, which allows external hardware to integrate seamlessly into the Kubernetes resource model. In addition to this traditional, static resource exposure, Kubernetes is evolving towards more dynamic allocation methods, notably through Dynamic Resource Allocation, offering greater flexibility and resource efficiency.

Kubernetes was designed from the ground up to be extensible. While CPUs and memory are natively supported, Kubernetes provides a standardized and extensible framework called Device Plugins for integrating specialized hardware resources such as GPUs, TPUs, and other accelerators.

Device Plugins allow vendors and Kubernetes operators to integrate custom hardware accelerators in a modular and pluggable manner. These plugins register with the Kubelet running on each node, advertising device availability and health status directly to Kubernetes, enabling resource-aware scheduling and workload isolation.

The Device Plugin interface supports a wide variety of specialized hardware, including Field-Programmable Gate Arrays (FPGAs), networking accelerators, storage

controllers, cryptographic modules, multimedia processors, and robotics hardware. Importantly for Generative AI, it also supports GPUs and other AI accelerators.

The Device Plugin mechanism operates through four core functions:

Device Discovery
: Plugins detect hardware devices on nodes and report their inventory to the Kubelet.

Resource Allocation
: When a workload requests specific hardware resources (e.g., GPUs), the device plugin handles exclusive allocation and sets up the runtime environment by exposing necessary device files or injecting environment variables.

Health Monitoring
: Plugins continuously monitor device health, ensuring Kubernetes is aware of unhealthy hardware to inform scheduling decisions.

Scheduler Integration
: Device plugins expose hardware as standard Kubernetes resources (e.g., `nvidia.com/gpu`). Pods request these resources explicitly in their resource declarations, ensuring accurate scheduling to nodes with available devices. We see more about resource-based scheduling in "Resource-based scheduling" on page 126.

Several well-known device plugins provide Kubernetes support for specific vendor hardware. Each of these plugins provides Kubernetes integration for the respective vendor's accelerators and enables workloads to consume GPU or TPU resources as first-class citizens within a Kubernetes cluster:

nvidia-device-plugin
: Official NVIDIA plugin exposing CUDA-enabled GPUs in Kubernetes, essential for GPU-accelerated AI workloads. We'll explore the NVIDIA device plugin further in the context of the NVIDIA GPU operator in "NVIDIA GPU Operator" on page 129.

amd-device-plugin
: Official AMD plugin integrating ROCm-based GPUs, suitable for high-performance computing and AI workloads.

intel-gpu-plugin
: Intel's plugin for integrated and discrete GPUs, enabling GPU resources in Kubernetes environments.

google-cloud-tpu-device-plugin
: Google's plugin for integrating Tensor Processing Units (TPUs), specialized hardware for TensorFlow workloads exclusively available in Google Kubernetes Engine (GKE).

Device plugins have become a foundational building block for GPU integration into Kubernetes. However, their approach to resource management has limitations. Devices are usually allocated exclusively to individual pods, which can lead to underutilized resources. Additionally, resource allocation is static, determined at scheduling time, making it less flexible for workloads with dynamic resource requirements.

To address these challenges, Kubernetes is evolving to support more dynamic allocation methods. A prominent advancement in this area is Dynamic Resource Allocation, which enables more flexible, real-time allocation and sharing of hardware resources, making it better suited for environments with diverse and changing workloads. We describe the current status of DRA in "Dynamic Resource Allocation" on page 127.

In the meantime, most production environments will continue to rely on the established resource request model for GPU scheduling.

Let's now take a closer how Kubernetes leverages both resource-based and label-based techniques to select the placement of GPU workloads in Kubernetes.

GPU Workload Scheduling

Kubernetes offers three complementary approaches to placing GPU-bound workloads. The first relies solely on numeric resource requests, while the second steers Pods with node labels and affinity rules. The third mechanism that influences automated placement for GPU workloads is based on Dynamic Resource Allocation (DRA), which is still experimental and in flux. We look at them separately so that the strengths - and the blind spots - of each method remain clear.

In the next section we discuss at label-based scheduling directly, show how selectors, affinity rules, and taints fill the precision gap, and discover the trade-offs of relying on labels for placement.

Label-based scheduling

When a cluster mixes several different kinds of GPUs, or when operators want to fence off GPU nodes from general workloads, labels become the steering wheel. Kubernetes offers three closely related mechanisms, each adding a different degree of control: `nodeSelector`, node affinity, and taints with tolerations.

nodeSelector

`nodeSelector` is the most direct approach. You attach a fixed label to every node that matches a certain characteristic and repeat that exact key - value pair in the Pod spec in the `nodeSelector` field. You can also leverage the labels that the NFD and GFD

operators attach to a node. See "GPU Discovery" on page 117 for what GPU-related labels are available for selection.

As seen in Example 5-3, the beauty of `nodeSelector` is its simplicity. A single line pins the workload to the desired node pool, with no extra scheduler overhead. That said, the rule is absolute: if the label is missing or misspelled, the Pod will never schedule. `nodeSelector` also cannot express alternatives; it is "T4 or nothing".

Example 5-3. Direct selection of a node with a node selector

```
apiVersion: v1
kind: Pod
metadata:
  name: t4-inference
spec:
  containers:
  - name: server
    image: myrepo/llm-server:latest
  nodeSelector:
    nvidia.com/gpu.product: Tesla-T4    ❶
```

❶ Select only nodes that are labelled for a Tesla T4 GPU

Node affinity

Node affinity builds on the same idea but allows richer expressions and soft preferences. Required terms act like an extended selector, while preferred terms let you nudge the scheduler toward the best node when several satisfy the hard constraints. Example 5-4 demonstrates this flexibility. This example relies on the labels added by the GFD described in "GPU Feature Discovery" on page 119.

Example 5-4. Node affinity for finer grained selections

```
apiVersion: v1
kind: Pod
metadata:
  name: a100-preferred
spec:
  containers:
  - name: llm
    image: myrepo/mt-server:latest
    resources:
      limits:
        nvidia.com/gpu: 4
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
```

```
      - matchExpressions:
        - key: nvidia.com/gpu.memory      ❶
          operator: Gt
          values: ["40000"]
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 1
      preference:
        matchExpressions:                 ❷
        - key: nvidia.com/gpu.family
          operator: In
          values: ["hopper"]
```

❶ Required conditions for a node for being considered as a scheduling target. In this example, the GPU needs at least 40 GiB memory.

❷ NVIDIA H100 are preferred, but will be ok, too if no H100 is available.

With affinity you can insist on a minimum memory size, then express a gentle preference for Hopper (H100) over Ampere (A100). If no Hopper node is free, the Pod still schedules on an Ampere node that meets the memory requirement. The downside is verbosity: long match expressions can clutter manifests, and too many hard clauses may unintentionally starve the workload.

Taints and tolerations

Sometimes operators want to flip the model and mark certain nodes as off-limits unless a Pod explicitly opts-in. A taint added by the administrator repels all Pods so that only those that carry a matching toleration are admitted.

In Example 5-5 a *taint key-value pair* `nvidia.com/gpu=true` is added with the *taint effect* `NoSchedule` which implies that no Pod that explicitly tolerates the `nvidia.com/gpu` constraint will be scheduled on all nodes that carry a label `nvidia.com/gpu.count`.

Example 5-5. Taint a node to be not considered for scheduling by default

```
# cluster-admin permission required
kubectl taint nodes -l nvidia.com/gpu.count nvidia.com/gpu=true:NoSchedule
```

Such a toleration is shown in Example 5-6, which allows it to be scheduled on nodes with a taint `nvidia.com/gpu` regardless of the value.

Example 5-6. Deployment with a toleration for nvidia.com/gpu taints

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```yaml
  name: gpu-serving
spec:
  replicas: 2
  template:
    spec:
      containers:
      - name: tgi
        image: ghcr.io/huggingface/tgi:latest
        resources:
          limits:                                  ❶
            nvidia.com/gpu: 1
      tolerations:
      - key: "nvidia.com/gpu"                       ❷
        operator: "Exists"
        effect: "NoSchedule"
```

❶ Require a `nvidia.com/gpu` resource as set by the NVIDIA device plugin. We explain resoure-based scheduling in "Resource-based scheduling" on page 126.

❷ Toleration that ignore any `nvidia.com/gpu` taints regardless of value, so that this Deployment can get also deployed on those nodes.

Taints are ideal for dedicating costly GPU nodes to GPU workloads or for cordoning nodes under maintenance. They work well in tandem with affinity or selectors: the taint keeps general Pods out, while affinity decides **which** GPU node is the best fit among those that remain.

More details about the various ways to influence Kubernetes' scheduling decissions can be found in the *Automated Placement* pattern in *Kubernetes Patterns*.

But which placement technique should be used in which context? It depends on the following:

- `nodeSelector` shines in small, homogeneous GPU fleets where a single label is enough.
- Node affinity becomes the tool of choice once you mix generations, memory sizes, or availability zones.
- Taints protect the GPU pool at cluster scope and pair naturally with the other two techniques for fine placement.

All three approaches share one limitation: they rely on static labels that administrators either maintain manually or are added by discovery operators like the NFD and GFD that we have described in "GPU Discovery" on page 117.

However, if you "just" want your Pod to run on a GPU-enabled node, there is an even easier way to specify your requirement, as we will see next.

Resource-based scheduling

The simplest way to schedule a GPU workload in Kubernetes is to declare the need for a GPU directly in the Pod specification. As soon as the NVIDIA device plugin is running on the nodes, it advertises every GPU as an extended resource - typically `nvidia.com/gpu`. A container that sets a limit like shown in Example 5-7 asks the Kubernetes scheduler to find a node with at least one free GPU.

Example 5-7. Require one NVIDIA GPU

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

The scheduler looks only at the numeric availability reported by the device plugin, binds the Pod to a qualifying node, and the kubelet then hands the container exclusive access to one of that node's GPUs. There are no extra labels to manage, no node selectors to remember, and no additional controllers to install. The mechanism is completely integrated with the familiar `requests` and `limits` resource model, so it feels like asking for CPU or memory - just with a different resource name.

That simplicity is its greatest strength. A single field in the Pod spec is enough to isolate the GPU at the device-file level, prevent other Pods from touching it, and let CUDA applications run without further configuration. Small clusters with one GPU type, or development environments where any GPU will do, rarely need more than this.

The downside is the lack of precision. All GPUs look identical to the scheduler, even if the cluster mixes V100s, A100s, or consumer-grade cards. A model that fits comfortably on an 80 GB A100 might not fit on a 16 GB T4, yet a plain `nvidia.com/gpu: 1` request treats them the same. There is also no built-in way to request a specific compute capability, restrict Pods to GPUs in Multi-Instance GPU mode, or ask for more than one GPU on a node with a particular interconnect topology.

In practice, teams work around this limitation by tagging nodes with additional labels, such as `gpu-type=A100`, and then combining the resource request with a `nodeSelector` or `nodeAffinity` rule to steer workloads to compatible hardware. It's a powerful tool, but it comes at the cost of extra coordination between the node inventory and the workload definitions.

A more elegant way to express nuanced requirements is possible through the Dynamic Resource Allocation mechanism described in "Dynamic Resource Allocation" on page 127. With DRA, you don't need to hard-code label conventions or rely on static resource requests. Instead, you declare what kind of GPU you want, and Kubernetes allocates a matching one dynamically.

Dynamic Resource Allocation

Dynamic Resource Allocation (DRA) is still under active development and may change in future Kubernetes releases. As of Kubernetes 1.33, it is available in beta but disabled by default.

The Kubernetes device plugin mechanism made it possible to expose GPUs and other specialized hardware as schedulable resources. With it, workloads can request accelerators like `nvidia.com/gpu` and have them allocated exclusively to a Pod at scheduling time. While this works well for many scenarios, it's a static model: devices are indistinguishable, fixed at scheduling time, and always allocated exclusively. In practice, this can lead to underutilized resources, coarse-grained scheduling decisions, and limitations when dealing with more advanced GPU configurations like Multi-Instance GPU or time-slicing that we discuss in "Sub-GPU Allocation" on page 132.

DRA is an effort to make device scheduling in Kubernetes more flexible, composable, and dynamic. Instead of tying device allocation directly to resource fields in the Pod specification, DRA introduces a new set of resource types. These abstractions shift the focus from "how many" devices to "what kind" of device a workload needs. The model is inspired by Kubernetes' volume provisioning, where users describe a desired resource and let the platform resolve it.

With DRA, workloads declare their device needs via ResourceClaimTemplate resources. These templates act as intent declarations that are resolved by the Kubernetes control plane and the installed DRA driver at scheduling time. This enables features that are difficult or impossible with static device plugins. For instance, a Pod can request a specific device class, like an A100 GPU with at least 40GB of memory, and the scheduler will only place the Pod on a node that can fulfill this requirement. The allocation happens just-in-time, allowing for smarter decisions and more efficient usage.

This flexibility becomes particularly useful when running LLM inference workloads that require specific GPU types, such as A100s with 80GB of memory. With DRA, it's possible to request exactly that - without relying on node labels, taints, or manual pod placement.

Example 5-8 illustrates this more advanced use case. We define a `ResourceClaimTemplate` that requests a GPU belonging to the A100 class, with at least 40 GiB of memory. We also define the allocation mode and count explicitly. This level of detail is something traditional resource requests cannot express.

Example 5-8. Template for a resource claim, usable by Deployments to add the claim to Pods that it creates

```
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: a100-claim-template
spec:
  spec:
    devices:
      requests:
        - name: high-memory-gpu
          deviceClassName: gpu.nvidia.com/a100  ❶
          allocationMode: ExactCount
          count: 1                              ❷
          parameters:
            minMemory: "40Gi"                   ❸
            migMode: "disabled"
```

❶ Requests a NVIDIA A100 GPU.

❷ One GPU required.

❸ GPU needs must have at least 40Gi memory, and Multi-Instance GPU mode has to be disabled.

In Example 5-8, we reference a logical device class called `gpu.nvidia.com/a100` - a level of abstraction that could include A100s with different configurations. We also ask for at least 40 GiB of memory and explicitly opt out of Multi-Instance GPU mode to ensure full GPU access.

The workload defined in Example 5-9 then references this template in its resource claims. The actual GPU allocation will only be made at scheduling time, and only if a matching device is available on one of the nodes.

Example 5-9.

```
apiVersion: batch/v1
kind: Deployment
metadata:
  name: inference-server
spec:
  template:
    spec:
      containers:
      - name: model-runner
        image: myorg/llm-inference:latest
        resources:
          claims:
```

```
      - name: high-memory-gpu                    ❶
  resourceClaims:
  - name: high-memory-gpu                        ❷
    resourceClaimTemplateName: a100-claim-template
```

❶ Reference to a resource claim that should be used when the Deployment creates one or more Pods.

❷ Reference to the resource claim template defined in Example 5-8.

The separation between the declaration and the actual allocation is what makes DRA so powerful. It opens the door for dynamic, demand-driven GPU provisioning - something that's difficult or impossible with traditional device plugins. It also allows drivers to perform more intelligent allocation strategies under the hood. Instead of a simple "pick the first available GPU", allocation can now consider current usage, power consumption, memory pressure, or other node-level constraints.

There are, however, a few caveats. DRA is not yet widely supported in production. As of mid 2025, it must be explicitly enabled via a feature gate, and the ecosystem around it is still catching up. The NVIDIA GPU DRA driver exists, but is marked experimental. Features like partial GPU requests, fine-grained Multi-Instance GPU partitioning, or topology-aware scheduling are still in development. Also, integration with cluster autoscalers or quota enforcement is limited.

Still, the potential is clear. DRA opens up Kubernetes to more intelligent GPU placement strategies, enables device sharing without resorting to hacks, and brings a more declarative mindset to accelerator provisioning. You no longer need to rely on static node labels, taints, or specialized resource counts to describe what your workload actually needs. Instead, you express the intent, and let Kubernetes and the driver handle the rest.

As DRA matures, it is likely to become the standard way of handling advanced hardware resources in Kubernetes. Vendors like NVIDIA are expected to integrate their device plugins, GPU operators, and runtime libraries with DRA to provide a seamless experience for both inference and training workloads.

Until DRA becomes production-grade, however, the combination of resource requests and label-based scheduling remains the standard approach for GPU scheduling.

NVIDIA GPU Operator

The previous sections showed how the Kubernetes Device Plugin exposes GPUs as schedulable resources ("Kubernetes GPU Device Plugins" on page 120) and how GFD enriches nodes with detailed NVIDIA-specific labels ("GPU Feature Discovery" on

page 119). The NVIDIA GPU Operator builds on and includes both pieces and adds everything else that is needed to run NVIDIA GPU workloads reliably in production. It installs drivers, container runtime hooks, monitoring agents, and offers two sharing mechanisms sub-GPU resource allocations in one declarative interface.

The operator automates the installation and configuration of all necessary components to enable GPU workloads to run efficiently and reliably.

The following components are part of the NVIDIA GPU operator:

NIVIDIA Drivers (Kernel Module and CUDA)
: At the heart of GPU enablement are the NVIDIA drivers - the kernel modules and user-space libraries that enable CUDA and GPU acceleration. The GPU Operator can deploy the official NVIDIA driver into each GPU node by running a privileged driver container. This container either compiles the driver for the node's kernel or fetches a precompiled driver if available. By containerizing driver installation, the operator ensures all GPU nodes have the required driver version without manual intervention.

GPU nodes should ideally run the same OS kernel version if you want to rely on the operator's driver container across them. Mixed OS versions might require pre-installing drivers manually. The operator's ClusterPolicy CR shown in Example 5-10 allows customizing the driver version or using precompiled binaries if needed.

GPU Feature Discovery
: We already covered the GFD in "GPU Feature Discovery" on page 119. The operator deploys GFD as a DaemonSet so that you don't have to install it manually on your own.

Kubernetes Device Plugin for GPUs
: The NVIDIA device plugin is deployed by the operator as a DaemonSet on GPU nodes. We already covered the Kubernetes Device plugin architecture in "Kubernetes GPU Device Plugins" on page 120 and how it introduces a new standard resource `nvidia.com/gpu` that can be used for resource-level scheduling, described in "Resource-based scheduling" on page 126. The NVIDIA Kubernetes Device plugins allows also for sub-GPU allocation as we will see later in "Sub-GPU Allocation" on page 132. The NVIDIA device plugin is a critical component; if it's not running or not working, your Pods will be stuck pending because Kubernetes doesn't think resources are available.

NVIDIA Container Toolkit (Runtime)
: In order for containers to actually use the GPU, they need the NVIDIA container runtime (which is part of the NVIDIA Container Toolkit). The GPU Operator deploys this toolkit on the nodes. The container runtime is essentially an exten-

sion to CRI-O and containerd that knows how to inject GPU drivers and libraries into containers when a GPU is requested. By managing the container runtime, the operator spares you from manually configuring the container runtime to include NVIDIA support. It's all handled automatically. The result is that any container which requests a GPU will have the necessary `/dev/nvidia0` device and CUDA libraries inside it, ready to execute GPU code, without the need to build special images.

Multi-Instance GPU (MIG) Manager
: On systems with MIG-capable GPUs (e.g., NVIDIA A100 or H100 cards), the operator includes a MIG Manager component. This service monitors the node's MIG configuration and can reconfigure the GPU's MIG partitions according to a desired state. By default it runs on MIG-capable nodes and will apply the MIG strategy you configure as we cover below when we describe the sharing options for NVIDIA GPUs. The MIG manager ensures that if a node should be in MIG mode with certain profiles carved out, it does so automatically on boot or when changes occur. Without it, an administrator would have to remote login into the node and use nvidia-smi to set up MIG partitions manually. The GPU Operator takes care of this, keeping MIG setups declarative.

GPU Monitoring with DCGM Exporter
: For completeness, the GPU Operator also typically deploys the Data Center GPU Manager (DCGM) Exporter as a DaemonSet. DCGM polls every GPU for utilisation, memory pressure, ECC errors, temperature, power draw, and a wealth of other counters, translating them into Prometheus metrics. Most users scrape the exporter with the cluster's Prometheus stack and surface graphs in Grafana. If you follow the observability guidance in Chapter 4, you already have everything in place; the operator merely wires the GPU side.

Operator configuraton with ClusterPolicy

The NVIDIA GPU operator is available for multiple Kubernetes distributions, including OpenShift, where it comes out-of-the-box as part of the OperatorHub catalog. It can also be installed on standard Kubernetes clusters using Helm charts or custom manifests provided by NVIDIA.

The configuration of the operator is done through the ClusterPolicy custom resource. This resource controls all aspects of the GPU operator components, including the device plugin configuration, enabling time slicing, and configuring MIG strategies. A ClusterPolicy can reference a custom ConfigMap to fine-tune the device plugin behavior, for example to enable time slicing or other GPU sharing mechanisms. Here you can specify a key from the ConfigMap that serves as a default if a GPU-enabled node is not labeled with a specific key of `nvidia.com/device-plugin.config=<key from configmap>`.

Example 5-10 shows a path to a custom ConfigMap for configuring the device plugin, enables the GPU Feature Discovery, and sets the MIG strategy to `mixed`.

Example 5-10. Example configuration for the NVIDIA GPU operator

```
apiVersion: nvidia.com/v1
kind: ClusterPolicy
metadata:
  name: gpu-cluster-policy
spec:
  gfd:                                 ❶
    enabled: true
  devicePlugin:
    config:                            ❷
      name: gpu-sharing-config
      default: sharing                 ❸
  mig:
    strategy: mixed                    ❹
```

❶ Enable the GPU Discovery Feature

❷ Point to the ConfigMap `gpu-sharing-config` that has extra configuration information, e.g. for configuring time slicing.

❸ The default references a key in the configmap. If set to an empty string, no default applies and nodes must be labeled with `nvidia.com/device-plugin.config=<config map key>` to pick up the decive plugin config.

❹ Set the MIG strategy to `mixed`. See below for more information about MIG.

Sub-GPU Allocation

The NVIDIA GPU Operator supports advanced GPU features for sharing a single GPU among multiple workloads. While sub-GPU resource allocation might not have the big relevance for operating LLMs, it's still important to understand to optimize your GPU usage.

The operator supports two modes, that can also be combined:

- **Time Slicing**: Allows multiple containers to share a GPU by allocating time-based slices.
- **MIG**: Available on certain GPUs (like A100 and H100) to partition a single GPU into isolated instances.

One of the powerful capabilities the NVIDIA GPU Operator enables is sharing a GPU among multiple workloads. This is especially relevant for AI inference or

serving multiple models, where you might not always need an entire GPU for each process. The operator supports two primary methods for GPU sharing: *Time-Slicing* and *MIG*. These can even be combined in certain cases.

Let's demystify these.

Time Slicing

By default, if a Pod requests a GPU (`nvidia.com/gpu: 1` in its resource requirements), Kubernetes gives it exclusive access to one physical GPU. Time-slicing is a feature that allows oversubscription of GPUs - i.e., advertising more than one virtual GPU per physical GPU, so that multiple Pods can be scheduled on the same GPU simultaneously. The NVIDIA device plugin (when configured accordingly) will create GPU replicas for each real GPU. For example, you could configure 1 physical GPU to be represented as 4 devices, allowing up to 4 Pods each to get what they think is one GPU, all actually running on the same hardware. These Pods' GPU workloads will interleave in time on the single GPU. Think of it like CPU time-sharing: if you have a 16-core CPU, you could schedule more than 16 CPU-bound processes by context switching - each gets a slice of time on the cores, but not all can run at full speed simultaneously. Similarly, with GPU time-slicing, if two Pods share one GPU, the GPU's scheduler will alternate between them, giving each a fair share of compute time.

GPU time-slicing is enabled by configuring the device plugin via a ConfigMap that is referenced in the operator's configuration like shown in Example 5-10. The corresponding ConfigMap is shown in Example 5-11.

Example 5-11. Example configuration for time-slicing

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: gpu-sharing-config
  namespace: gpu-operator-resources
data:
  sharing: |
    version: v1
    sharing:
      timeslicing:
        renameByDefault: true       ❶
        resources:
        - name: nvidia.com/gpu
          replicas: 8               ❷
```

❶ Use `nvidia.com/gpu.shared` instead of `nvidia.com/gpu` if enabled for the resource specification. This makes it easier to depict or avoid shared GPU instances.

❷ Overcommit level, i.e. offed 8 virtual GPUs for each physical GPU.

We specify which resource name to use (e.g. `nvidia.com/gpu` or `nvidia.com/gpu.shared`) and a replicas count. By setting `nvidia.com/gpu` with `replicas: 8`, each physical GPU is exposed as 8 schedulable units. Consequently, the node's `nvidia.com/gpu` label will display the total virtual GPU count (e.g., 8 for one GPU, 80 for ten). A `gpu.replicas=8` label will also be added to the node to signify this oversubscription. Optionally you can rename the shared resource to `nvidia.com/gpu.shared` to distinguish it from truly exclusive GPUs if you set the option `renameByDefault` to `true`.

Unlike MIG, time-slicing does not provide memory or fault isolation between the Pods sharing the GPU. All Pods on the same physical GPU have access to the entire GPU memory and are in the same fault domain. This means if one process crashes the GPU (like an illegal memory access causing a GPU reset), it will affect the other workloads on that GPU as well. Also, if one Pod grabs most of the GPU's memory, the others may fail to allocate memory. Time-slicing only guarantees a share of compute time, not memory quotas.

A Pod requesting multiple time-sliced GPUs (e.g., `nvidia.com/gpu: 2` when GPUs are in shared mode) doesn't get 2x a single GPU - it would likely end up on two physical GPUs that are each shared with others, which is usually not what you want. In fact, by default the device plugin can be set to fail such requests >1 to avoid confusion. Generally, time-slicing is intended for scenarios where each Pod uses only 1 GPU (which is actually a share of a real GPU). For multi-GPU workloads, you'd typically keep those GPUs exclusive or use MIG.

Time-slicing is great for oversubscribing GPUs in environments where workloads are bursty or lightweight. For example, if you have a powerful GPU but lots of small inference tasks or interactive notebooks, you could run several on one GPU so it's utilized more efficiently. Each job might run a bit slower if they truly contend for GPU, but overall throughput can improve. Time-slicing also allows older GPUs (which don't support MIG) to be shared. If you have NVIDIA T4 or V100 GPUs in a lab cluster running many smaller models, time-slicing might let you run 2-3 per GPU concurrently, maximizing hardware use. Just be mindful of the lack of memory isolation: you must size your models so that together they fit in GPU RAM. Admittedly, time-slicing might be not so useful in the context of LLMs which are typically so large that they need to allocate the full physical memory offered by a GPU.

Multi-Instance GPU

MIG is a feature available on NVIDIA Ampere and newer GPUs (like A100, A30, H100) that allows you to partition a physical GPU into several hardware-isolated instances. Each instance (or *MIG slice*) has its own dedicated compute cores, memory carve-out, and even separate engine contexts - it's like having multiple smaller GPUs in one card. For example, an A100 40GB GPU can be split into up to 7 MIG instances, the smallest being 1g.5gb (one GPU slice with 5 GB memory each). Each MIG device acts like a mini GPU with guaranteed memory and isolated Streaming Multiprocessor (SM) resources.

The NVIDIA device plugin can expose MIG partitions as schedulable resources in two ways:

Single MIG Strategy
: All MIG instances on a node are advertised under the generic `nvidia.com/gpu` resource (just like normal GPUs). This strategy assumes each GPU is identically partitioned. For instance, if every A100 on the node is split into 7x 5GB instances, then a node with 2 A100s would report nvidia.com/gpu: 14. When a Pod requests 1 GPU, it actually gets one MIG slice (5 GB). The node labels (`gpu.product`, `gpu.count`, etc.) are adjusted to reflect MIG (you'd see `gpu.product = ...-MIG-1g.5gb` and `gpu.count = 14` in the example). This approach keeps things simple for users, but it requires homogeneous MIG setup on all GPUs.

Mixed MIG Strategy
: MIG instances are exposed as distinct resource types, named by their MIG profile (e.g., `nvidia.com/mig-1g.5gb`, `nvidia.com/mig-4g.20gb`, etc.). In this mode, a node with an A100 might advertise several different resources if it has a mix of MIG sizes. A user can request a specific MIG size by using the corresponding resource name in the Pod spec. For example, a Pod could request `nvidia.com/mig-2g.10gb: 1` to get a roughly 10 GB MIG instance. This strategy is more flexible (GPUs in a node could be split differently or even remain whole), but it's a bit more advanced to schedule since users need to know which MIG type to ask for.

In both cases, the GPU Operator's MIG manager will handle creating the MIG partitions on the GPU as specified by the `mig.strategy` in the ClusterPolicy (either `single` or `mixed`) as shown in Example 5-10. If MIG mode is off (`none` strategy), then GPUs are not partitioned at all.

Unlike time-slicing, MIG provides strong isolation. Each MIG instance has a fixed fraction of the GPU's memory - it cannot use more than its allocation, which prevents one workload from stealing the memory of the others. Fault isolation is also improved: if one MIG instance crashes or resets, the others can continue unaffected. This makes MIG attractive for multi-tenant or production scenarios where you want

to safely run different applications on the same physical GPU. This is ideal if each model service only needs a few GBs of GPU memory.

The trade-off is granularity and overhead. You are limited to the MIG profiles defined by NVIDIA (you can't create an arbitrary 6 GB slice, only the fixed sizes offered by the card). Also, if one job could have used the whole GPU at times, MIG partitions mean it's hard-limited to its share - there's no concept of borrowing unused capacity from others. In contrast, time-slicing could let one Pod burst to use the whole GPU if the others are idle (because nothing prevents it from grabbing more memory or compute when available, whereas MIG would keep it confined). Therefore, MIG is best when you have fairly steady parallel uses for the GPU that each fit in a partition. For LLM and generative AI workloads, MIG is particularly useful for inference serving scenarios or running many smaller experiments. If you have a large model (that needs >40GB, for instance), MIG won't help - you need the full GPU or multiple GPUs. But if you're hosting multiple smaller models (say 7 different language models each requiring ~5 GB), MIG can be very helpful, effectively giving each model its own *virtual* GPU with guaranteed memory.

Interestingly, time-slicing and MIG aren't mutually exclusive. You can time-slice MIG instances too. For example, you could split a GPU into 2 MIG instances, and then oversubscribe each MIG instance 2x with time-slicing. This would present 4 schedulable units per GPU. This is advanced and only needed in corner cases - but the operator does support it (it will append `-SHARED` to MIG device product labels if both are enabled). One might do this if they want the memory isolation of MIG (say two big slices), but also want to occasionally run two Pods in one slice. However, for most, it's either MIG or time-shared, not both simultaneously, due to complexity in managing performance.

To summarize the difference, MIG partitions a GPU into smaller dedicated slices (each with fixed memory and compute capacity), whereas time-slicing treats the whole GPU as a single pool that multiple jobs take turns using, sharing time but not guaranteeing memory splits. MIG gives you isolation and predictability, while time-slicing gives you flexibility and potentially higher utilization if not all jobs are busy at once. For LLM training (which often consumes entire GPUs or multiple GPUs), typically neither MIG nor time-slicing is used - you just allocate GPUs exclusively. But for LLM inference and related workloads (fine-tuning smaller models, running many experiments, serving many models), these sharing techniques are very useful. A common pattern is to use MIG for strict multi-tenancy or production QA tests, and use time-slicing in dev environments or for oversubscribing on less critical batch jobs where you don't mind if they slow each other down.

`nvidia-smi` is NVIDIA's System Management Interface tool, providing real-time monitoring and management of NVIDIA GPU devices. It offers insights into GPU utilization, memory usage, temperature, power consumption, and active processes. By executing `nvidia-smi`, users can obtain a snapshot of the current state of all GPUs in the system. For continuous monitoring, the `-l` flag can be used to refresh the output at specified intervals (e.g., `nvidia-smi -l 5` updates every 5 seconds). This tool is invaluable for diagnosing performance issues, ensuring that GPUs are operating within optimal parameters, and verifying that applications are utilizing GPU resources as intended. Additionally, it aids in detecting anomalies such as thermal throttling or unexpected memory consumption, facilitating proactive troubleshooting in GPU-accelerated environments.

You can it easily run it directly with `kubectl`, too:

```
patch=$(cat <<EOT
[{
  "op":"add",
  "path":"/spec/containers/0/resources",
  "value":{"limits":{"nvidia.com/gpu":1}}
}]
EOT
)

kubectl run --rm -it gpu-pod \
  --image=nvidia/cuda:12.8.1-base-ubi9 \
  --restart=Never \
  --overrides=$patch --override-type=json -- nvidia-smi
```

Sub-GPU techniques like time slicing or multi-instane GPUs are useful for squeezing many small or mid-sized models onto the same card, but large-language models rarely fall into that category. In practice the bottleneck is not how to split one GPU - it is that even the biggest card is still too small. So turn the perspective upside down and look at multi-GPU inference: stitching several GPUs, and sometimes several nodes, into a single serving engine capable of hosting today's heavyweight LLMs.

Multi-GPU Inference

In the previous section we learned how to divide a single physical GPU among many workloads. For large-language-model serving the inverse situation is far more common. Running inference for large language models (LLMs) often demands more GPU memory and compute than a single GPU can provide.

There are two primary motives for using multiple GPUs in LLM inference: *model parallelism* and *throughput scaling* (also known as *data parallelism*).

Model parallelism is required when a single model is too large to fit into one GPU's memory - the model is split across GPUs so that each GPU holds part of the model and collectively they handle one inference request. In contrast, *throughput scaling* uses multiple GPUs to host replicas of the same model, serving different requests in parallel to increase overall queries-per-second (QPS).

Let's look at the various types of multi-GPU usage separately.

Throughput Scaling

One motive for multi-GPU inference is to increase the overall throughput. In this scenario, each GPU (or each group of GPUs) runs an independent replica of the model to serve different requests concurrently. This is essentially data parallelism at the inference level - it does not accelerate any single query's latency, but it allows more queries to be processed in parallel, thereby boosting QPS. For example, if you have 4 GPUs and a moderate-sized LLM that fits in one GPU, you can either: deploy 4 instances of the model (each on its own GPU) to handle 4x the traffic, or use a multi-threaded server that automatically dispatches requests to each GPU. Kubernetes makes it easy to scale this way by running multiple replica pods, each requesting one GPU and serving the model behind a load balancer service.

Figure 5-1 shows how to fan out to multiple replicase of the same model for many concurrent requests.

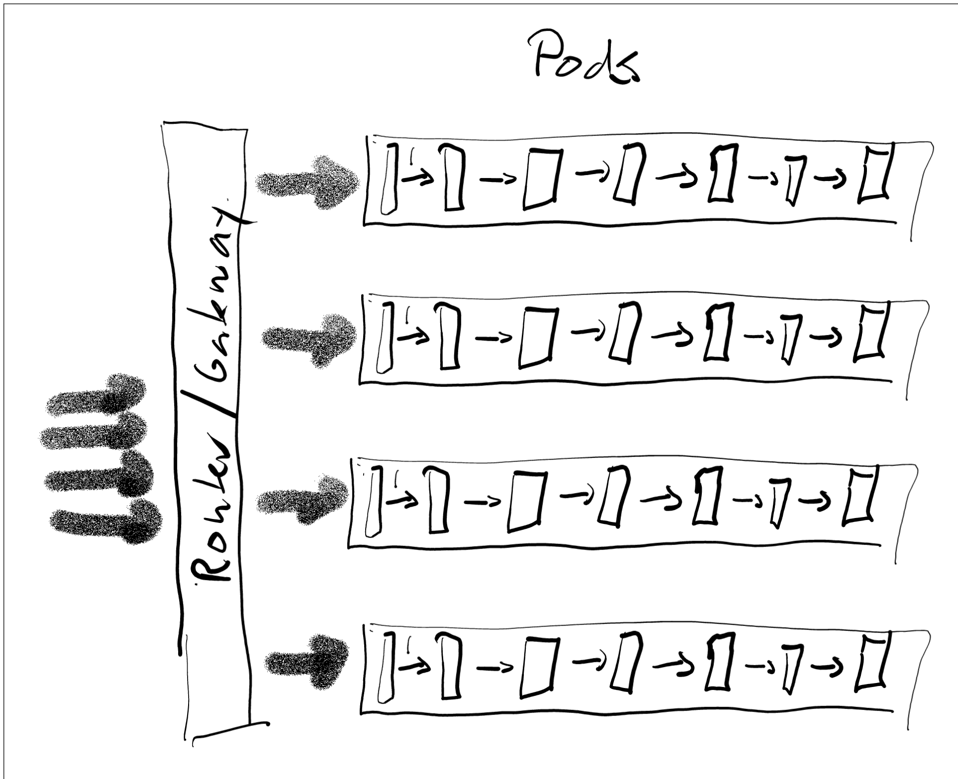


Figure 5-1. Throughput Scaling

Throughput scaling is ideal when you need to serve many simultaneous users or API requests and the model fits in a single GPU's memory. For instance, an LLM with 7B parameters can often be quantized to ~8 GB, fitting on a 16 GB GPU - you might run 8 replicas on 8 GPUs to handle lots of chats in parallel. The limitation is that this does nothing to reduce the latency of a single query. Each query is still processed by one GPU from start to finish (if the model is not parallelized), so if you have a single very large request that one GPU would take 10 seconds to handle, adding more GPUs for throughput scaling won't speed that one request up (it will just allow handling other requests concurrently). In fact, serving a single request on multiple model replicas would be wasteful - instead, model parallelism as described in the next section is needed for that case. Another limitation is resource usage: running *N* replicas means storing *N* copies of the model weights in memory. This can be inefficient if the model is large and memory is limited. Some frameworks support multi-stream batching on a single model instance to improve utilization (e.g. vLLM can dynamically batch multiple incoming queries on one GPU to improve throughput), which is an alternative to full replication. In summary, throughput scaling via multiple GPUs is straightforward (often just horizontal scaling of pods),

but be mindful that it increases memory footprint linearly and saturates overall GPU compute only if you indeed have enough concurrent load. If request rate is low, those extra GPUs might sit idle - in which case, one might consolidate work onto fewer GPUs or even share GPUs among multiple models via time-slicing or MIG as we have described in "Sub-GPU Allocation" on page 132.

Model Parallelism

The second motive for multi-GPU inference to allow a single large model to be served by multiple GPUs in unison. This is called *model parallelism* and is necessary for modern LLMs with tens or hundreds of billions of parameters that exceed the memory of one GPU. The model is partitioned so that each GPU holds a portion of the network and computes part of the forward pass.

The two most common approaches to model parallelism are *tensor parallelism* and *pipeline parallelism*, which can also be combined for very large deployments. Model parallelism focuses on reducing per-GPU memory usage and potentially latency for one inference by leveraging multiple devices in parallel, at the cost of added communication between GPUs. High-bandwidth interconnects (like NVIDIA NVLink or NVSwitch) are often critical here to handle the frequent data exchanges without bottlenecks.

Tensor Parallelism

Tensor parallelism slices the computations within each layer across multiple GPUs. In this scheme, each GPU holds a shard of the layer's weights (for example, splitting a large weight matrix by columns or rows) and processes a portion of the layer's input. GPUs then exchange partial results to construct the full output of the layer. For instance, if a fully-connected layer has a weight matrix too large for one GPU, it can be divided into multiple slices - each GPU multiplies the input by its slice, and the partial outputs are concatenated or summed to form the complete output. This approach keeps all GPUs busy on the same layer (improving per-token latency) and effectively multiplies the available memory bandwidth by using several GPUs in parallel. Tensor parallelism directly reduces the memory burden per GPU, allowing extremely large models to load (e.g. splitting a 70B-parameter model across 2-4 GPUs means each holds only 35B-17.5B params).

It also can yield lower latency per token since GPUs compute in parallel. However, TP introduces frequent communication overhead - GPUs must sync after processing each layer or attention head. If the interconnect is not fast enough, communication can dominate runtime (in poorly partitioned cases, communication can consume 50-70% of inference time).

For this reason, tensor parallelism is best confined to single-node setups with high-bandwidth links (PCIe with NVLink or NVSwitch). In fact, it's not advisable to do

fine-grained TP across multiple nodes with standard networking due to latency costs. We talk in more detail about single-node versus multi-node setup for GPU usage later in "Single-Node versus Multi-Node inference" on page 143. In practice, the maximum tensor parallel degree is often the number of GPUs in one server (e.g. 4-way TP on a 4-GPU node). Beyond that, one should either use a machine with more GPUs or switch to pipeline parallelism between nodes.

Pipeline Parallelism

Pipeline parallelism splits the model vertically by layers, assigning different consecutive layers to different GPUs. In this approach, the input sequence is processed by the first few layers on GPU0, after which the intermediate activations are passed to GPU1 for the subsequent layers, and so on - resembling an assembly line. Pipeline parallelism thus requires storing and transferring the intermediate activations at pipeline stages, but not every layer's outputs as in tensor parallelism - communications happen only once per pipeline stage (per forward pass) rather than at every layer operation. The key advantage of pipeline parallelism is that it minimizes inter-GPU communication frequency. Each activation hand-off is a one-time message per stage, making pipeline parallelism more tolerant of slower interconnects or multi-node environments. This makes pipeline parallelism ideal for scenarios where GPUs are across different servers (connected by Ethernet or InfiniBand) or when NVLink/NVSwitch isn't available. It allows scaling to models that exceed even a multi-GPU node's total memory (e.g. sharding a 175B model across 2 nodes). However, pipeline parallelism does not improve single-request latency - in fact, it can increase latency due to its sequential stage processing. A pipeline also introduces idle time because the next GPU in the pipeline cannot start processing the next token's data until the previous GPU has finished the previous token, etc. Thus, unless carefully managed, multiple GPUs in a naive pipeline might be underutilized. To mitigate this, frameworks use micro-batching or scheduling techniques: splitting the incoming batch or sequence into micro-batches that are fed in staggered fashion so all pipeline stages stay busy in parallel. For example, NVIDIA's FasterTransformer and vLLM implement pipelining with automated micro-batch scheduling to avoid idle times. The bottom line is that pipeline parallelism shines for multi-node scaling and high-throughput batch processing (where latency of individual queries is less critical).

Figure 5-2 illustrates the different sharding techniques of tensor and pipeline parallelism.

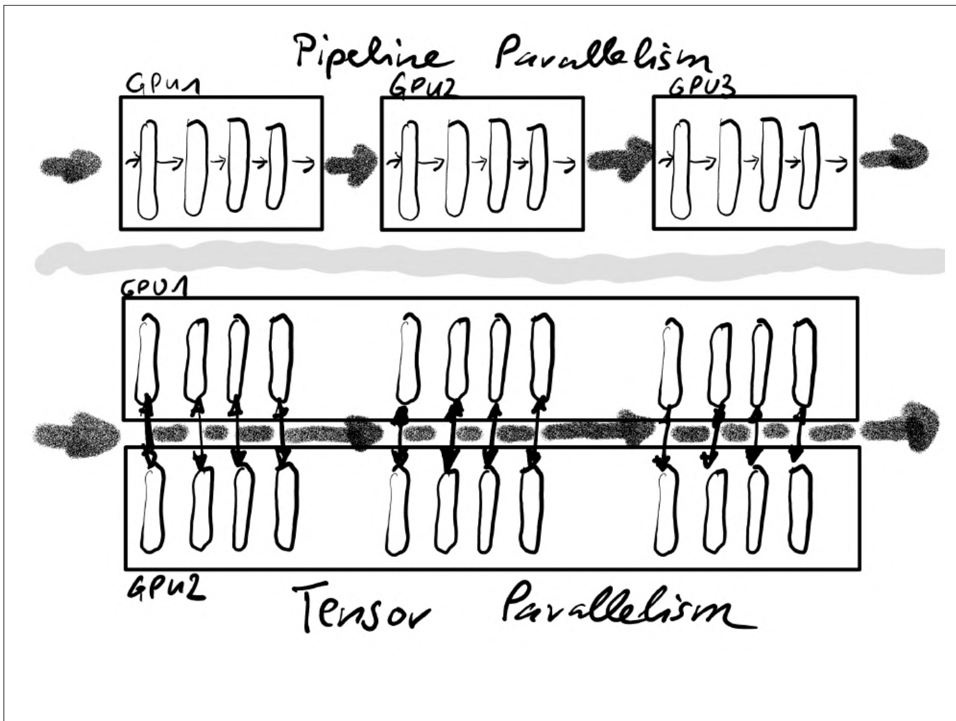


Figure 5-2. Tensor Parallelism and Pipeline Parallelism

These two strategies can be combined for maximum scalability. Many systems adopt a hybrid parallelism approach: using tensor parallelism within each node and pipeline parallelism across nodes. This hybrid approach leverages fast local links for intra-node splitting, and uses pipeline stages to span multiple machines without requiring excessive cross-node communication. The rule of thumb is: use pipeline parallelism across nodes and tensor parallelism within a node when network links are slow, but if you have a very fast interconnect between nodes (e.g. Infiniband or NVLink bridge), tensor parallelism can extend across nodes as well.

In all cases, distributed inference requires coordination - GPUs must communicate intermediate results using collective operations (all-reduce, all-gather, send/recv, etc.), typically via NVIDIA's NCCL library over high-speed links. This added complexity means there is some efficiency loss versus single-GPU operation, but it enables serving models of unprecedented size. It's also worth noting that if one GPU and the node in a model-parallel group fails, the inference will fail; there isn't graceful fault tolerance for a partially missing model shard. Thus, deploying model parallel inference in Kubernetes may benefit from pod affinity/anti-affinity rules (to co-locate GPUs or separate failure domains) and appropriate health checks to restart the whole group if one part dies. In practice, many teams keep model-parallel deployments to

a single node when possible for simpler failure handling and use multi-node only for truly massive models.

Let's take a closer look when multi node inference makes actual sense and what its challenges are.

Single-Node versus Multi-Node inference

When deploying on Kubernetes, you may have nodes equipped with multiple GPUs, and you might also consider spreading a model across GPUs on different nodes. There are important distinctions in these scenarios.

Single-node multi-GPU inference means all GPUs used for the model or replicas are in the same server. As already mentioned, this has an advantage of high-speed local interconnects. Within one machine, GPUs often communicate via PCIe (and if it's a high-end GPU-enabled server, via NVLink or NVSwitch between GPUs). For instance, NVIDIA DGX-class nodes have NVSwitch connecting all 8 GPUs with ~600 GB/s bandwidth, which makes intra-node communication far faster than typical network links. As a result, parallel strategies that involve frequent communication (like tensor parallelism) work very well within a single node. In Kubernetes, using multiple GPUs on one node is straightforward: you request the number of `nvidia.com/gpu` resources in the Pod spec and the scheduler will place the pod on a node that has that many free GPUs. All GPUs assigned to a pod will be visible to the container (e.g. as as an environment variable `CUDA_VISIBLE_DEVICES`). Your inference server or code can then initialize model parallelism across those devices.

Figure 5-3 shows how multiple GPUs can be quickly accessed on a single node.

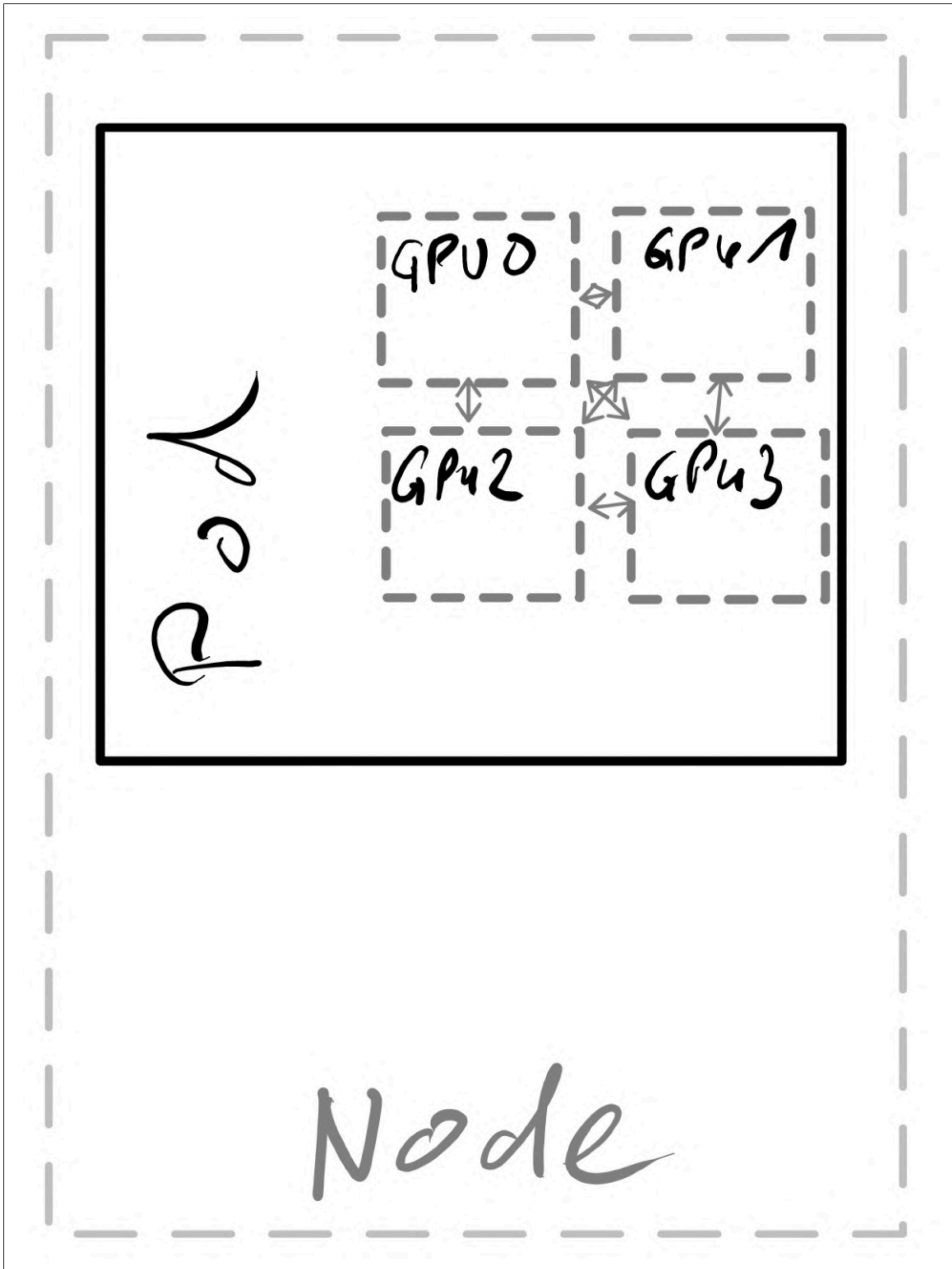


Figure 5-3. Multiple GPUs on a single node

By contrast, *multi-node multi-GPU inference* involves splitting the model or the serving load across GPUs that reside on different machines. This is necessary when

the model is so large that no single node has enough GPUs or GPU memory (for example, some teams have run 175B+ parameter models across two or more 8xA100 nodes). In this case, the communication will go over the network interface between nodes like InfiniBand or Ethernet. The bandwidth is typically much lower and latency higher than on intra-node links. For example, a 100 Gbit Ethernet (12.5 GB/s) is an order of magnitude slower than NVLink (~600 GB/s) - meaning that a tensor-parallel all-reduce that might be negligible on NVLink could become a bottleneck across the network. Therefore, pipeline parallelism (which, as noted, sends larger chunks less frequently) often becomes the preferred strategy across nodes.

With pipeline parallelism, each node processes a substantial portion of the workload before passing it to the next, making the system more resilient to network latency. Additionally, if multi-node is used, it's recommended to use the fastest network available and to ensure NCCL is configured to use RDMA if possible. NCCL can operate over sockets or InfiniBand; in Kubernetes, one must also ensure the pods can discover each other's addresses for NCCL (sometimes using Kubernetes service IPs or host networking for performance).

What is NCCL and RDMA ?

NCCL (NVIDIA Collective Communication Library) is a high-performance library designed for efficient GPU-to-GPU communication in multi-GPU and multi-node environments. It provides optimized collective communication primitives such as *all-reduce*, *broadcast*, *reduce-scatter*, and *all-gather*, which are essential for synchronizing model parameters or intermediate results during tensor and pipeline parallelism. NCCL is typically not used directly by end-users - instead, it is leveraged under the hood by inference runtimes like vLLM, and frameworks such as PyTorch, which abstract its complexity behind higher-level APIs. However, in distributed Kubernetes deployments, advanced users may need to tune NCCL-related settings (e.g., `NCCL_SOCKET_IFNAME`) to ensure optimal performance over specific network interfaces. When available, RDMA (Remote Direct Memory Access) can be used by NCCL to bypass the CPU and access memory directly between nodes, significantly reducing latency and improving bandwidth in multi-node inference setups. Properly configured, NCCL with RDMA plays a crucial role in achieving scalable, high-throughput inference for large language models across multiple GPUs and nodes.

Kubernetes does not natively schedule a single pod across multiple nodes, so multi-node inference typically means running one pod per node and coordinating them externally. Popular inference runtimes often leverage orchestration frameworks to simplify this process. For instance, vLLM internally uses Ray.io to orchestrate multi-node inference transparently, managing pod-to-pod communication, node discovery, and fault tolerance. Other runtimes, such as Hugging Face's TGI, rely on Kubernetes-native constructs like StatefulSets or Deployments, where one pod acts as a coordina-

tor (commonly referred to as “rank 0”) and manages communication between model partitions on different pods.

In Figure 5-4 multiple nodes with each hosting multiple GPUs are orchestrated by vLLM for inference.

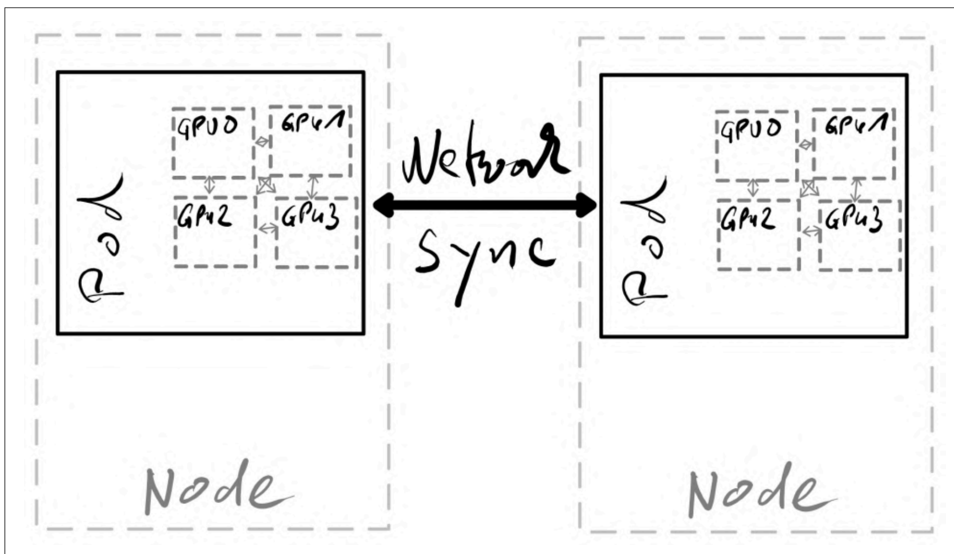


Figure 5-4. Multiple GPUs on a multiple nodes

Regardless of the orchestration method chosen, managing pod affinity (ensuring pods land on distinct GPU nodes or optimized locations) and service discovery (letting pods resolve each other by name or IP addresses) is essential. Kubernetes provides built-in mechanisms, such as Pod hostnames and subdomains, to facilitate pod discovery. Additionally, NCCL, used by inference runtimes for efficient GPU communication, can perform topology discovery automatically within a node but typically requires explicit network interface configuration across nodes.

Latency and bandwidth differences mean that the scaling efficiency (throughput or speed-up) going from single-node to multi-node may drop. It’s common to see near-linear speed-up within one node (e.g. 4 GPUs giving ~3.5x throughput of 1 GPU for a well-optimized model) but going multi-node might give diminishing returns if network becomes a bottleneck. Also, collective operations (like all-reduce) across nodes need to be synchronized - if one node is slightly slower or its network latency is higher, it can slow down the others. This makes performance less predictable, and also means the slowest node dictates the pace (e.g. if one node’s GPU is busy with some other task or GC pause, it could stall the pipeline).

In single-node multi-GPU inference, the failure of the node naturally leads to pod termination, which Kubernetes handles straightforwardly. In contrast, multi-node

model-parallel inference involves significant complexity, as a failure of any participating pod typically disrupts the entire inference job due to incomplete model partitions. Recovery usually entails restarting the full group of pods. Kubernetes concepts like PodDisruptionBudgets help minimize disruptions during planned maintenance. Some advanced setups may consider checkpointing strategies to mitigate such failures, although these techniques are less common for stateless inference workloads and more often used during post training.

In summary, single-node deployments remain preferable for model-parallel inference due to lower complexity and higher efficiency. However, when multi-node deployments are required due to model size, pipeline parallelism or other network-efficient methods, combined with robust orchestration solutions like Ray.io or Kubernetes-native deployment patterns, ensure reliable and efficient large-scale inference operations.

GPU Resource Optimizations

Maximizing GPU utilization and avoiding memory waste is key in production LLM inference, since GPUs are expensive resources. Here are some best practices and considerations for optimizing GPU usage in a Kubernetes environment:

GPU Memory Defragmentation
: As models load and unload, or as dynamic inference workloads allocate memory (for example, varying sequence lengths), the GPU's memory allocator can become fragmented - meaning free memory is in many small chunks rather than one contiguous block. This can prevent large models from being loaded or lead to out-of-memory errors even when enough total memory is free (just not contiguously). It's often best to pre-allocate large blocks (e.g. load all model weights on startup, use memory pools for scratch space) to avoid heap fragmentation. Frameworks like PyTorch have a caching allocator that helps, but long-running pods might still suffer fragmentation over time. If you notice GPU memory increasing or OOMs after many requests, a strategy is to periodically restart the pod to clear fragmentation - or utilize advanced allocators. Research is exploring GPU memory defragmentation at runtime, but those aren't yet in mainstream use. On the inference side, vLLM's PagedAttention is essentially a defragmentation technique for the KV cache. In order to detect memory fragmentation, a proper monitoring setup is essential. If you see that after serving many requests the available memory decreases its probably because of memory defragmentation.

GPU Sharing and Consolidation
: Aim to keep GPUs busy - an idle GPU is wasted money. If your LLM only uses 30% of a GPU's compute and memory, consider running multiple model instances or other workloads on the same GPU. This can be done with MIG

on supported hardware for clear separation. For example, you might run two 6B-parameter models on one 80GB A100, each in a 40GB MIG slice. If MIG isn't available, and you trust the workloads to play nice, you could enable GPU time-slicing to let two pods share one GPU. For more details see "Sub-GPU Allocation" on page 132. Another approach is to run a multi-model server that loads several models onto one GPU and routes requests (if they all fit in memory). Some serving frameworks (like NVIDIA Triton or Multi-Model Server) support multiple models per GPU, dynamically unloading less-used models if needed. The best practice is to profile your usage: if a model only uses say 50% of GPU memory, that remaining 50% could host another smaller model or a second copy to double throughput. Just be cautious to not overload memory - leave some margin since driver overhead and fragmentation can eat a few percent. Kubernetes doesn't natively know if a GPU is "only half used" - it's up to you to bin-pack wisely using MIG or by deploying multiple pods to the same node.

Quantization and Compilation

Optimizing the model itself can reduce GPU needs. Techniques like 4-bit or 8-bit quantization of weights dramatically cut memory per model copy (at some accuracy cost). If an LLM can be quantized from 16-bit to 8-bit with negligible quality loss, you potentially halve the number of GPUs needed. Many open models have 8-bit or 4-bit quantized versions available. These allow, for instance, a 70B model to fit on a single 48GB GPU in 4-bit mode (70B * 4 bytes/param → 280GB, vs 70B * 0.5 byte/param → ~35GB in 4-bit). The vLLM and TGI servers support loading such quantized models. Additionally, consider using optimized runtimes to improve inference speed per GPU. Faster models mean you can handle more load with the same hardware, improving utilization.

Auto-Scale

Autoscaling multi-GPU deployments can be tricky if they are model-parallel (you can't just half-replica a single model that's split across 4 GPUs - you either remove the whole group or add another whole group). But for throughput scaling, Kubernetes-based autoscaling (like with KEDA, the Horizontal Pod Autoscaled (HPA) or Knative) on RPS, concurrency or latency is effective. Details to the Kubernetes intrinsic autoscaling can be found in the *Elastic Scale* pattern from "Kubernetes Patterns".

Placement and Affinity

For multi-GPU nodes it is important to know the topology. On some 8-GPU servers, not all GPUs are directly connected - there might be NVLink links in a mesh or groups, e.g. a NVIDIA DGX A100 has NVSwitch all-to-all, but other systems might have 2 groups of 4 GPUs each. If your model parallelism uses 4 GPUs, you might get better performance if those 4 are all within one NVLink group. You can use `nvidia-smi topo -m` to show the mesh grouping. Kubernetes won't automatically account for that, but you can use the Node's hardware knowl-

edge and assign specific GPU indices by using the Device Plugin's capabilities to pick specific GPUs by index. This is an advanced optimization - for most cases, Kubernetes will just assign any 4 GPUs, but if you care about intra-node latency, you might pin to say GPU0-3 if they're on the same NVSwitch cluster.

Optimize I/O and Initialization
: Large models take time to load from disk or network into GPU memory. If you scale pods up and down, you pay that cost each time. Amortize it by keeping pods warm if possible (if you can afford it). We describe optimized model loading techniques in Chapter 3.

Monitor GPU Health
: GPUs can encounter issues like ECC memory errors or high temperature throttling. Ensure you have node-level monitoring and alerts for such events. Kubernetes won't automatically reschedule a pod if the GPU starts erroring but hasn't crashed - you might need a daemon that checks with `nvidia-smi` for errors and then taints the node or restarts pods. Running the NVIDIA DCGM (Data Center GPU Manager) and integrating with Kubernetes node health can help. A flaky GPU in a model-parallel group can cause wrong results or crashes, so it's important to catch hardware issues quickly.

By following these best practices - defragmenting memory, smartly sharing GPUs, leveraging scaling and optimization tools, and closely monitoring - you can achieve high utilization and reliability for multi-GPU LLM inference on Kubernetes.

Lessons learned

Kubernetes wasn't born with GPU workloads in mind, but it has grown up quickly. Through device plugins, feature discovery, and vendor operators, the platform can now expose GPU resources in a structured and declarative way. Still, it's up to us to make effective use of those building blocks.

We've seen how GPU workloads differ from CPU-bound ones - not just in terms of resources, but also in lifecycle, scheduling sensitivity, and placement constraints. Concepts like time slicing or MIG help squeeze more out of your hardware, but require orchestration and support from the underlying stack. Just enabling GPUs isn't enough. You also need to understand how your workloads behave, what kind of sharing they tolerate, and what kind of isolation they need.

Things get even more complex when you scale up. If a single GPU can't hold your model, you'll need to coordinate multiple of them. This introduces model parallelism strategies like tensor and pipeline parallelism, which demand coordination across pods, nodes, and sometimes even across runtimes. Kubernetes gives you the primitives but you have to wire the rest together.

At the end of the day, running GPU workloads on Kubernetes is not a magic trick. It's a layering of well-understood patterns - feature discovery, device scheduling, topology awareness, and workload orchestration.

Kubernetes continues to evolve to meet the demands of GPU-intensive workloads. The emerging DRA API is one example of that trajectory - moving toward more flexible, pluggable, and workload-aware resource handling. As generative models grow in size and complexity, expect more such innovations in Kubernetes to simplify and streamline GPU integration.

CHAPTER 6

Running In Production

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

By now, you have likely deployed your first Large Language Model (LLMs) to run on Kubernetes. It responds to requests, maybe even with decent latency. But production isn't about working once - it's about working consistently, at scale, and under load.

This chapter is all about that transition. We'll dive into what it takes to make LLM inference stable and efficient in real-world scenarios. That includes topics you might expect - like parameter tuning - but also those that are easy to overlook, like runtime memory planning, routing sticky requests to cache-warm replicas, choosing when (and how) to compress your model, and advanced topologies that requires dedicated network configuration.

It's tempting to treat a model server like any other container. Just set a few resource limits, expose a Service, and call it a day. But GenAI workloads stretch Kubernetes in new ways. You'll see how to make the most of the platform you already know, while avoiding the traps that can quietly erode performance and burn through your GPU budget.

In the next sections, we'll look at how to evaluate and optimize the model itself, how to make better use of limited GPU capacity, and how to benchmark and route traffic for maximum throughput and minimal cost. We'll also discuss emerging tools and APIs that help with smarter autoscaling and request routing - especially when your setup goes beyond a single replica.

Let's start with the most fundamental decision: how to pick and tune a model that matches your use case without wasting compute cycles.

Model and runtime tuning

Probably the most important aspect to consider when a team approaches the development of the first real application based on Generative AI is the selection of the model to use. Most teams begin with a managed service like OpenAI's ChatGPT where configuration options are limited to configure but there are many situations where on-premise infrastructure is a must-have and the selection of the model at that point is critical. This selection is based on many different factors like type of task, type of workload (i.e. real time vs batch inference) and number of concurrent requests.

The size of the model matters but two models with the same size might have different model architectures and training techniques with the consequence that the results of the same query can go from very accurate to completely wrong.

The number of models available is very high and new models are published so it is important to follow some core principles to help the model selection phase.

There is no silver bullet or *single model to rule them all* but this doesn't mean that the selection process should consider all the models available on Hugging Face! One of the common tasks performed during the development of a Predictive AI model includes checking and comparing models, trained for the same task, based on their accuracy, so it should be useful to find one or more metrics to compare the accuracy of LLM.

If a traditional Predictive AI model is trained to provide prediction of a very specific problem, on the other side a LLM is trained on a very vast set of data and it has different skills to perform multiple tasks, so it is first necessary to identify the task that is critical for the application and based on that, select one or more accuracy metrics to compare the models.

This phase is critical to guide the selection of the model and base it on specific metrics instead of (limited) manual testing. The problem is very complex and there is an entire research field about language model evaluation.

Language model evaluation

The evaluation of a language model can target many different aspects like measure how knowledgeable a model is, or the ability of a model to produce content without toxic language or even how good a model is with reasoning tasks. This definition is not specific to LLMs, there are many traditional language models defined before the LLM area where the same principle applies.

One of the most important application of language evaluation is to verify model safety related with specific tasks to measure model toxicity or robustness.

There are many different projects that provide one or more evaluation benchmarks, one of the most used suite is EleutherAI's lm-evaluation-harness that includes more than 100 out-of-the-box tasks. There are other libraries too and new evaluation techniques are often defined to test the models in scenarios that are more and more complex.

Traditionally an evaluation task includes a dataset with a set of inputs / outputs to use (usually as multi-choice questionnaires to simplify the analysis) and an evaluation function to compute the metric. This format makes it very easy for a subject matter expert to review the dataset and it can be easily grouped in subtopics to more easily categorize the abilities of the model.

Each benchmark is essentially a procedure that invokes the target model with a set of predefined questions/answers and analyzes the results so it takes time (even hours) to run it and the model must be deployed so it can be very expensive to perform. Fortunately for all the most used models it is possible to find online leaderboards that collect the results of the evaluation for multiple benchmarks, allowing for easy comparison..

A leaderboard is a table where different models are compared using one or more metrics but at the same time the usage of similar leaderboards has security and trust implications because there is no way to verify if the published numbers are true without locally re-executing the test. The general advice is to use the leaderboards to perform the initial assessment to produce a short list of models and then perform additional analysis.

A good starting point is the leaderboard page on the Hugging Face website that links many of them grouped by categories like mathematics abilities or safety of the model. See Figure 6-1 to get more details on the execution flow of an evaluation request.

Given that each task can takes multiple minutes to perform, it is usually performed asynchronously inside a pipeline/job, for example TrustyAI project provides a wrapper of `lm-evaluation-harness` library introducing the `LMEvalJob CRD` to performs the task and check the status to be notified when it has been completed. A full example can be found in TrustyAI LM-Eval documentation.

In addition to the hundreds of existing evaluation tasks, it is also possible to create a custom one but this prevents the possibility to easily compare different models or a different version of the models using online leaderboards.

One of the most used benchmarks is Measuring Massive Multitask Language Understanding (MMLU) because its dataset covers a large set of multiple-choice questions from various branches of knowledge that is grouped by tasks including a very large set of topics: abstract algebra, high school european history, high school government, politics, and much more. The same approach has been extended to cover multimodal models by Massive Multi-discipline Multimodal Understanding (MMMU).

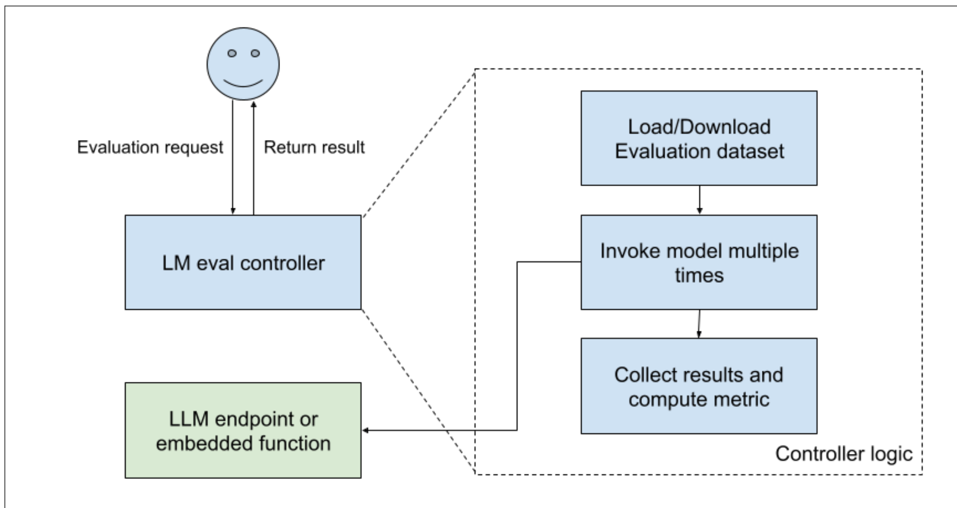


Figure 6-1. Language model evaluation execution flow

The selection of the model is a key aspect and it happens at a very early stage of the project but the concept to evaluate a model and compare the metrics is not only useful during model selection. The initial model selection is usually performed by the Data Scientist / AI Engineer but it is critical to optimize the model before running it at scale: a model can be compressed using quantization techniques or it is even possible to collect production data and use them to distill the model in a smaller one.

Both these compression techniques are changing the weights of the model impacting the overall accuracy of the model so it is necessary to perform the evaluation again to measure the impact. The application of similar techniques can produce a model that is less than half of the original size increasing by 2-3 times the throughput of the runtime on the same hardware with essentially the same accuracy of the original model. When properly applied the compressed model can recover more than 99% of the original accuracy.

Language model compression

A LLM like `Meta-Llama-3.1-8B-Instruct` has 8 billion (8B) parameters, this means that there are 8 billion floating point values and each of them is used as weight (multiplier) for the corresponding input of the neural network.

Each floating point value is represented with 16 bits so the minimal memory requirement to load the model is 15 GB (16 bit x 8 billion). S neural network might have numerous layers (say, 32), with each neuron's output feeding into the subsequent layer increasing the overall memory footprint of the model. This is called activation and it is another 16 bit floating point value.

Finally, both the KV cache values and the intermediate output results are represented using 16-bit precision. Go back to section "Understanding LLM" on page 88 for more details on this topic.

All these floating point numbers concur to the total memory footprint of a LLM for the memory of the GPU. The technique to compress the model is called quantization.

Quantization

Quantization is the name of a set of techniques that aims to reduce this footprint using a less precise representation, like an 8 bit floating point (FP8) or an integer representation (INT8). These techniques go beyond simply rounding floating points or reducing decimals. Instead, they involve mechanisms specifically designed to compensate for errors during execution.

Compressing the model is only half of the solution because if the runtime doesn't have native quantization support the compressed model will be converted back to 16 bit precision during the processing to evaluate the neural network and produce the activations. The sweet spot of quantization is a runtime that has optimized kernel implementations (GPU functions) that are able to process quantized data end to end. This is where the scalability magic happens!

The vLLM runtime has native support for most of the quantization techniques both on the compression side with the LLM compressor project but especially on the runtime side with multiple optimized kernels that are automatically enabled when the runtime recognize that the model is quantized.

This area of research looks very promising to make LLM serving more cost effective, potentially becoming the future standard for models and runtimes, but some critical aspects remain. First of all, the specialized kernels that are critical to properly scale quantized models are hardware specific so not all GPUs support it yet and in general it is possible to have a dramatic accuracy drop when the quantization goes wrong.

Although the first aspect, ensuring comprehensive GPU coverage, may simply require time, the second aspect demands meticulous control. The compression is not lossless so it is critical to evaluate the quality of the model after the compression to make sure the level of accuracy has not been impacted too much.

There are multiple examples of models which have been quantized to use 8 bit per floating point (FP8) that have recovered more than 99% of the accuracy of the original model and the library `llmcompressor` simplifies the whole procedure with built-in calibration. The usage of the library is quite straightforward but it requires some knowledge to understand the different parameters (Example 6-1).

Example 6-1. Compress a LLM using llmcompressor

```
from llmcompressor.modifiers.quantization import GPTQModifier
from llmcompressor.modifiers.smoothquant import SmoothQuantModifier
from llmcompressor.transformers import oneshot

# Select quantization algorithm. In this case, we:
#   * apply SmoothQuant to make the activations easier to quantize
#   * quantize the weights to int8 with GPTQ (static per channel)
#   * quantize the activations to int8 (dynamic per token)
recipe = [                                                          ❶
    SmoothQuantModifier(smoothing_strength=0.8),
    GPTQModifier(scheme="W8A8", targets="Linear", ignore=["lm_head"]),  ❷
]
# Apply quantization using the built in open_platypus dataset.
oneshot(
    model="TinyLlama/TinyLlama-1.1B-Chat-v1.0",
    dataset="open_platypus",                                        ❸
    recipe=recipe,
    output_dir="TinyLlama-1.1B-Chat-v1.0-INT8",                     ❹
    max_seq_length=2048,
    num_calibration_samples=512,
)
```

❶ The `recipe` defines the quantization pipeline to be applied with one or more techniques.

❷ The `scheme` W8A8 means that the weight is quantized with 8 bits and same for activations.

❸ It is necessary to specify a dataset to be used to calibrate the quantization

❹ The output directory will contain the compressed model with all the configuration files (`config.json`, `tokenizer.json`, etc.) necessary to serve the model so that the vLLM runtime knows how to load the model without any additional/special parameter.

A good practice is to compute the accuracy metrics of the original model before deployment to production, using language model evaluation techniques. Furthermore, integrating compression directly into the MLOps/LLMOps pipeline can provide an optimized starting point for the system without requiring custom configuration.

Model performance benchmark

Chapter 4 covered how to observe a LLM and which metrics are critical to track usage and system responsiveness in particular for realtime use cases like a chatbot. It's crucial to react quickly to resolve latency issues by properly configuring scaling, rate limiting, or even rejecting new requests with appropriate error messages.

Regardless of the mitigation action that will be decided to apply to the system, the prerequisite is to measure the overall behavior of the system under different conditions and with different workloads.

Performance and capacity testing is not something new to software development, every mature release pipeline should include it and a served model from an end-to-end perspective is very similar: it is an endpoint that accepts requests and generate tests.

Various tools are available for performance testings allowing you to invoke an endpoint. A runtime can definitely be tested with traditional load generators. This is how LLM performance testing has been done, and it is still sometimes used. Over time more specialized tools have been developed that are able to compute more specialized metrics and mix performance testing and model evaluation combining different scenarios and datasets.

The LLM community is very active and there are many different tools that are available some of the most comprehensive are:

GuideLLM
: The GuideLLM project has been created specifically as a tool to evaluate a model and optimize the deployment of LLMs. It simulates different types of workloads to be equivalent to a real-world scenario, GuideLLM can be used as interactive tool to gauge performance and used resources with a specific hardware configuration. The tool covers a large set of rate scenarios that it is possible to configure, from synchronous scenario, where every request is chained one after the other, to more advanced scenarios with fixed concurrency or even Poisson distribution with the mean at the specified rate. After the benchmark, GuideLLM produces a report with the distribution of the latency and many other useful information (Example 6-2).

Example 6-2. Run a benchmark with GuideLLM

```
guidellm benchmark \
  --target http://127.0.0.1:8000 \                          ❶
  --model mistralai/Mixtral-8x7B-Instruct-v0.1 \
  --output-path output_file.json \
  --rate-type sweep \                                       ❷
  --data "prompt_tokens=256,output_tokens_min=128" \        ❸
  --max-seconds 400
  --warmup-percent 0.2


# Example of output report after the execution
# The results can be parsed as JSON in the output file
 ─ Benchmarks ───────────────────────────────────────────────────────────────────────
│ [1...]   synchronous   Req:    0.3 req/s,    2.89s Lat,      1.0 Conc,      138 Comp,   … │
│                        Tok:   88.0 gen/s,   450.6 tot/s,  135.2ms TTFT,   10.9ms ITL,   … │
│ [1...]   throughput    Req:    8.6 req/s,   55.44s Lat,    476.0 Conc,     3427 Comp,   … │
│                        Tok: 2193.7 gen/s, 10955.5 tot/s, 27421.5ms TTFT,  109.9ms ITL… │
│ [1...]   constant      Req:    1.5 req/s,    8.43s Lat,     12.4 Conc,      587 Comp,   … │
│                        Tok:  373.1 gen/s,  1865.1 tot/s,  112.4ms TTFT,   32.9ms ITL,   … │
│ [1...]   constant      Req:    2.7 req/s,    9.93s Lat,     26.7 Conc,     1075 Comp,   … │
│                        Tok:  690.1 gen/s,  3437.7 tot/s,  114.0ms TTFT,   38.4ms ITL,   … │
│ [1...]   constant      Req:    3.7 req/s,   11.75s Lat,     43.1 Conc,     1459 Comp,   … │
│                        Tok:  941.0 gen/s,  4677.1 tot/s,  122.5ms TTFT,   45.5ms ITL,   … │
│ [1...]   constant      Req:    4.9 req/s,   13.23s Lat,     64.2 Conc,     1940 Comp,   … │
│                        Tok: 1243.3 gen/s,  6196.8 tot/s,  127.5ms TTFT,   51.4ms ITL,   … │
│ [1...]   constant      Req:    6.0 req/s,   17.90s Lat,    107.1 Conc,     2392 Comp,   … │
│                        Tok: 1529.5 gen/s,  7646.4 tot/s,  145.6ms TTFT,   69.7ms ITL,   … │
│ [1...]   constant      Req:    6.9 req/s,   21.68s Lat,    148.8 Conc,     2743 Comp,   … │
│                        Tok: 1754.8 gen/s,  8736.1 tot/s,  192.5ms TTFT,   84.4ms ITL,   … │
│ [1...]   constant      Req:    7.9 req/s,   29.53s Lat,    232.8 Conc,     3151 Comp,   … │
│                        Tok: 2015.5 gen/s, 10062.3 tot/s,  400.6ms TTFT,  114.4ms ITL, … │
│ [1...]   constant      Req:    8.2 req/s,   41.59s Lat,    341.0 Conc,     3278 Comp,   … │
│                        Tok: 2092.4 gen/s, 10451.5 tot/s, 12899.4ms TTFT,  112.9ms ITL… │
 ────────────────────────────────────────────────────────────────────────────────────
```

❶ The target model to test must be deployed before running the test.

❷ GuideLLM supports various rate types, with "sweep" as the default. This versatile option covers multiple scenarios, from sequences of requests to constant rates, providing baseline system numbers without needing a specific workload definition.

❸ To ensure relevant testing, it's important to specify input and output sizes that match what's expected in production. The tool defaults to using a local copy of *Pride and Prejudice*, but it also supports datasets from Hugging Face or custom local datasets.

MLPerf Inference

The challenge to properly test performance of machine learning models is not new nor limited to LLMs and indeed there are communities where companies, individual contributors and academy are collaborating to define tools and publish results. MLCommons is an AI engineering consortium, built on a philosophy of open collaboration to improve AI systems. This organization provides many different tools, one of them is MLPerf Inference and it has been extended over time to support LLMs. The results are published periodically on MLCommons website spitting data center configurations (published at MLPerf Inference: Datacenter page) and EDGE/device (published at MLPerf Inference: Edge page).

Inference Perf

Inference Perf is a GenAI inference performance benchmarking tool proposed and incubated by the Kubernetes WG-Serving group and sponsored by Kubernetes SIG Scalability. It is a community effort that includes the support from different companies. It is specialized for Generative AI and it is possible to specify arbitrary dataset to be used to simulate a scenario that is similar to a real world situation. The library can run locally or it can deploy in a cluster connecting to previously deployed a model with the assumption that it exposes OpenAI compatible API.

vLLM benchmark suite

Performance is a critical aspect for inference engines like vLLM, which provides publicly available nightly benchmark jobs with instructions. It is not a proper tool but it is more a set of scripts that can be used to measure the performance of a specific model or hardware configuration.

As mentioned before, while other traditional load generators can be used, this would require implementing the LLM-specific metrics like Time to First Token (TTFT) and Inter-Token Latency (ITL). Whatever is the tool that has been selected to measure the performance, the integration in the CI/CD system is usually quite straightforward, it is enough to create a task that deploys the model and the perform the test using the tool. The most critical aspect is to use an environment that is equivalent to the target/production cluster, especially regarding the model and the number of GPUs where the model is deployed.

Integrating the performance test in the release pipeline and storing the result is critical to properly size the cluster and learn the capacity limits. These numbers are critical for the capacity planning but also to configure rate limiting / API gateway.

vLLM runtime parameters tuning

The section "Understanding LLM" on page 88 explained how the inference of LLM works, which metrics are important to be monitored and finally how important KV cache is to make LLM serving efficient. The model is now compressed, ready to be

deployed and, thanks to the performance tests, there are also data related to overall system performance that can be used to guide the tuning of the runtime.

This section is specific to vLLM runtime but most of the content applies to other LLM runtimes too.

The development of vLLM project is very active, new optimizations and new models are added on a weekly basis. Improved defaults are implemented in every release so in most cases it is not necessary to tune it and *it just works*. The only aspects that vLLM cannot easily infer automatically are the type of workload and the hardware assigned to be used and this is where it is possible to help the engine with the configuration.

Although the default configuration of vLLM is usually a valid starting point, it is necessary to perform some sizing analysis to calculate how much VRAM memory the GPU needs to have.

How to calculate model memory requirements

There are many different factors that contribute to the definition of the memory requirements, most of them are related to the model architecture and the model size together with the number of concurrent requests.

The main driver of the memory requirements of a model is the number of parameters: a model with 8 billion (8B) parameters requires way less memory compared to very large models that can reach more than 400 billion of parameters.

Each parameter in a full-size model typically occupies 2 bytes (16 bits, `float16` or `bfloat16`) of memory. Through compression techniques, this can be reduced to 1 byte (8 bits, `FP8` or `INT8`) per parameter. Multiplying the number of parameters with the size of each them produces the baseline requirement of memory.

In addition to this baseline there is some infrastructure space related to the optimized kernels loaded to the GPU that usually is between 300 MB and 2 GB.

The cost of the activations, that represent the intermediate status during the execution, should be also considered and it is directly related with the hidden size value and the number of layers. Both these values can be found in the `config.json` file of the model (i.e. 40 layers). Activation memory requirements might be limited like 200-300 MB when the sequence length is small (i.e. 512) but this changes dramatically with larger sequences because its cost grows quadratically.

Finally, the generation of the output requires the output tensor that has a cost that is directly related with the size of the vocabulary, the sequence length and the batch size.

Let's do a full example with a 8B model that uses parameters with `float16` format, 2048 context size and batch size 1: the baseline memory requirement is about 15 GB (8 billion x 2 bytes), it is usually necessary to consider about 1 GB of additional

memory for the infrastructure, activation space is about 900 MB (assuming 4096 as hidden size) and finally about 400 MB for the output layer.

The total VRAM requirement is about 17.3 GB, which seems reasonable. However, simply increasing the batch size to 10 for processing multiple requests simultaneously almost doubles this requirement, pushing it to over 28 GB of VRAM.

This topic is much larger than this simplified example, new techniques and model architectures are defined frequently changing the memory implications, for example there are ongoing evolution like the Mamba LLM architecture that should reduce dramatically the size requirements for the KV cache.

There are online tools that can help this calculation like TitanML's Model Memory Calculator but to learn more, including all the formulas that have been applied, there are different online articles. More details can be found in this blogpost from Alexander Smirnov and in the EleutherAI's blog, both articles are highly recommended.

The vLLM runtime greedily utilizes available resources to maximize throughput. Therefore, having more resources directly improves performance. In a perfect scenario vLLM has enough GPU available to load the model in the GPU VRAM but also it has enough space for the activation (the intermediate results of each layer of the neural network) and for the KV cache so it is never necessary to evict data from the cache that are still necessary thus the runtime has to compute them again.

When this is not the case the first symptom is going to be a higher inter token latency and implicitly less throughput. This is not the only way to detect it because vLLM is usually explicit in the logs when there are similar scenarios. We already looked at startup logs from vLLM in the previous chapter in Example 4-1 but let's now focus on memory information (Example 6-3).

Example 6-3. vLLM logs information about memory

```
...
INFO [model_runner.py:1097] Loading model weights took 14.9888 GB                ❶
INFO [worker.py:241] Memory profiling takes 0.67 seconds
INFO [worker.py:241] the current vLLM instance can use total_gpu_memory (79.14GiB)  ❷
        x gpu_memory_utilization (0.90) = 71.22GiB
INFO [worker.py:241] model weights take 14.99GiB; non_torch_memory takes 0.12GiB;   ❸
        PyTorch activation peak memory takes 1.19GiB; the rest of the memory     ❹
        reserved for KV Cache is 54.93GiB.                                       ❺
...
WARNING [scheduler.py:1057] Sequence group 0 is preempted by PreemptionMode.SWAP mode ❻
        because there is not enough KV cache space. This can affect the end-to-end
        performance. Increase gpu_memory_utilization or tensor_parallel_size to
        provide more KV cache memory. total_cumulative_preemption_cnt=1
```

1. During the startup of the vLLM, after the model is loaded in the memory of the GPU, the log includes the size of the weights.
2. There is a very useful log entry that explains the total memory of the GPU that vLLM can use.
3. In addition of the model weights there is some additional memory used by the engine
4. Activation takes some memory too: the value of a "peak value" depends on how many nodes of the neural network are activated during an execution. For example with Mixture of Experts model architecture, only a subset of the model is activated every time.
5. The rest of the memory is assigned to the KV cache.
6. During the execution of the model, this log is produced when the KV cache memory is not enough so vLLM has to swap some of the value outside the VRAM.

The log explicitly reports the information when the size allocated to the KV cache is not big enough, a single entry of this log is probably not critical but when this happens multiple times it is probably better to consider applying some tuning. Fortunately the log is very detailed so the log message includes some suggestions to try to mitigate/overcome the problem. Let's now describe the parameters that it is important to consider:

`gpu-memory-utilization`
: The parameter's default value is `0.9`, indicating the percentage of available memory vLLM can use. This setting controls vLLM's memory consumption, a crucial feature given past issues where it could exceed assigned GPU memory, risking Out-Of-Memory (OOM) errors. While `0.9` serves as a safety threshold, vLLM's stability has improved. Consequently, it's now often safe to increase this value closer to `1.0`, allowing access to that extra 10% of memory.

`max-model-len`
: This parameter is crucial and must be configured based on the specific LLM use case or task. The KV cache size directly corresponds to the context size (input prompt plus generated text). While tuning this value directly impacts memory usage, an overly aggressive limit on context size could result in responses lacking enough tokens to address the use case. For instance, RAG patterns often require a substantial context.

`max-num-seqs` *or* `max-num-batched-tokens`
: The vLLM engine batches the input to increase GPU usage and increase the throughput. This might have some latency implication but in a highly concurrent production scenarios the impact is usually very limited. At the same time the bigger the batch size is, the higher is the KV cache space so it is possible to reduce this value to save some memory.

`tensor-parallel-size`
: Unlike previous parameters, increasing tensor parallel size necessitates additional hardware. This process shards the model weights, providing more available memory for the KV cache on each GPU. While this requires multiple GPUs, cross-GPU communication within the same node is generally not a bottleneck due to dedicated high-speed interfaces.

`pipeline-parallel-size`
: Pipeline parallelism distributes the model layers across multiple GPUs, whereas tensor parallelism splits individual tensors. Both approaches are compatible and increase available KV cache memory, but pipeline parallelism is more commonly used for inference.

`data-parallel-size`
: Similar to pipeline parallelism, this approach splits data into parallel groups, enabling multi-node serving across GPUs on different servers. This technique is applicable when a cluster contains more than one GPU-equipped server. While it increases available KV cache memory, it introduces the complexity of a multi-node scenario, which requires dedicated connectivity.

`cpu-offload-gb`
: The last parameter in this list offloads part of the model to the CPU, allowing for the deployment of models larger than the available GPU memory. While seemingly useful for KV cache management, this option significantly impacts throughput. It is strongly discouraged for production due to a massive performance drop and the loss of critical GPU-specific optimizations (e.g., specialized kernels), which are essential for efficient LLM serving.

Using these parameters and knowing the characteristics of the workload is critical to tune the runtime to maximize the performance and matching the expected service level expectation. When this is not enough it is possible to add more GPUs to the system but it is also possible to consider additional optimizations thanks to Kubernetes stack and in particular on the networking part as described in "LLM-aware routing" on page 170.

Autoscaling

Even if the model has been properly compressed and the runtime parameters properly tuned this doesn't mean that the setup is perfect to get the most out of the hardware resources that are available, actually it is still far from that scenario. The workload that a single instance of the runtime is able to handle, even when properly tuned, is limited and in a real world production scenario, it is very likely that a single replica per model is not enough to handle the workload during its peak.

In a realtime inference scenario the end user latency is critical so the metrics Time To First Token (TTFT) and Inter Token Latency (ITL) should be monitored to guide the routing and the scaling while in case of offline inference it is more important to tune the batch size and the throughput.

Fortunately Kubernetes has native support for horizontal pod autoscaling dynamically balancing the requests across the different replicas so scaling the workload should not be a concern, right? As usual LLM workloads raise some specific challenges that requires the Kubernetes ecosystem itself to evolve. Let's now cover the main options for autoscaling that can be adopted.

Horizontal Pod Autoscaler (HPA)
: While the default HPA works out-of-the-box and requires no additional dependencies, it primarily monitors CPU and memory. This makes it less suitable for LLM workloads, which predominantly impact GPUs, thus requiring a more flexible autoscaling solution.

Knative Pod Autoscale (KPA)
: The Knative Serving project offers KPA, a more flexible autoscaler that bases its decisions on the number of requests. Its default "stable" mode uses a time-based window for concurrency calculations to scale. A "panic" mode is also available, employing a much shorter window for faster reactions to workload changes. While KPA is a better fit for LLMs than HPA and integrates natively with KServe (using Serverless deploymentMode), it was designed for microservices where Pod scaling is rapid. Unfortunately, LLMs are complex, GPU-dependent deployments that can take many minutes to load based on model size. This makes dynamic workload scaling less practical without significant tuning (refer to "Optimize vLLM startup time" on page 167). More importantly, a request-based approach like KPA's fails to consider that the number of requests doesn't directly correlate with the runtime workload: one request could generate many tokens, while the next generates only a few.

KEDA (Kubernetes Event-driven Autoscaling)
: KEDA is a project that has been created to scale event-driven workloads where the request is not a HTTP request but a message from (usually) a queue. The challenge that KEDA creators had to solve is similar to the challenge of LLM

scaling: there are many different technologies to implement event driven architectures and it is necessary to have a flexible API to configure how to retrieve the information to measure the overall pressure on the system. The solution that has been implemented enables the possibility to configure a query based on metrics to guide the scaling. This is because every queue system publish indicators like the number of messages in the queue to be processed but every technology has a different approach (i.e. push based vs poll based) and different naming conventions. This flexibility works well with vLLM too because the runtime publishes metrics like `vllm:num_requests_waiting` that measure how many requests are still waiting to be processed or `vllm:time_to_first_token_seconds/vllm:time_per_output_token_seconds` to keep track of the time to produce every token. KServe natively supports KPA using Serverless deploymentMode but it also supports KEDA via the RawDeployment deploymentMode. Go back to "KServe" on page 38 to have more information on the different deployment modes of KServe. The configuration of KEDA autoscaler is done in the `InferenceService` specification with the definition of the query to perform and it supports the possibility to fetch directly from PodMetric or from an external source. Both options are equivalent if the configured query is limited to vLLM metrics that are local to the Pod, in this scenario fetching metrics directly from the Pod reduce the latency in the autoscaler making it more responsive. The external source option is more flexible because it is possible to collect metrics from different sources (even different replicas of vLLM) and performs more advanced query that joins all information (Example 6-4).

Example 6-4. Example of KServe and KEDA

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: Meta-Llama-3-8B
  annotations:
    serving.kserve.io/deploymentMode: RawDeployment          ❶
    serving.kserve.io/autoscalerClass: "keda"                ❷
    sidecar.opentelemetry.io/inject: "Meta-Llama-3-8B"       ❸
spec:
  predictor:
    model:
      modelFormat:
        name: huggingface
      args:
        - --model_name=llama3
        - --model_id=meta-llama/meta-llama-3-8b
    minReplicas: 1
    maxReplicas: 5
    autoScaling:
      metrics:
```

```
        - type: PodMetric                                    ❹
          podmetric:
            metric:
              backend: "opentelemetry"                       ❺
              metricNames:
                - vllm:num_requests_running
              query: "vllm:num_requests_running"             ❻
            target:
              type: Value
              value: "4"                                     ❼
#        - type: External                                    ❽
#          external:
#            metric:
#              backend: "prometheus"
#              serverAddress: "http://prometheus.url:9092"   ❾
#              query: "vllm:num_requests_running"
```

❶ RawDeployment is required to use KEDA autoscaler, Serverless uses Knative which has its own autoscaler

❷ This annotation enables KEDA autoscaler

❸ One option to collect metrics local to the Pod is via OpenTelemetry sidecar collector

❹ PodMetric type configures KEDA to query directly the Pod to collect the metrics

❺ It is necessary to specify the backend because OpenTelemetry and Prometheus has some differences in the protocol

❻ The query that KEDA performs to make the decision can be a single value or a more complex query following the PromQL syntax

❼ In this example the value 4 means that KEDA will increase the number of replica (within the defined 1-5 boundaries) if there are at least 4 request already running in vLLM based on some preliminary benchmark that has measured latency distribution with a higher number of concurrent requests

❽ This commented spec is the other configuration where the metrics are queried from an external source

❾ When the type is External it is necessary to specify the server address where the query should be executed

The evolution of LLM serving includes the autoscaling topic, new more custom techniques are under development for most complex scenarios like disaggregated

prefill. Disaggregated prefill is described in "Disaggregated Serving" on page 177 but it is very different as a concept compared to the traditional Kubernetes autoscaler definition becoming more like a "runtime controller" that observes the status of the system and automatically rebalance the role of the different replicas or can decide to reduce or increase the number or replicas for each role.

In this section we covered the challenges to configure a proper autoscaler strategy for LLM deployments but when there are multiple replicas another challenge requires attention, tune load balancing strategy. If the load balancer is not LLM-aware it can produce a suboptimal distribution of the request impacting the stability of the latency of the system. This challenge is described and addressed in "LLM-aware routing" on page 170.

Optimize vLLM startup time

In "Autoscaling" on page 164 we learned how to introduce a more specialized autoscaler configuration for LLMs but in a real world scenario dynamic scaling is not really applicable if the time to start a new replica is many minutes.

The size of LLMs stretches the core design principles behind Kubernetes itself, some of the biggest LLMs can requires almost 1TB of storage just to store the model itself, the name really represents them… They are *large*!

This aspect makes it hard to configure an efficient autoscaling strategy that can detect a peak of workload and scale the deployment on the fly. There is a physical bottleneck in the time to transfer a similar amount of data.

Performing proper capacity planning together with performance and scalability tests is in general a good practice before deploying an application in production and this general advice is even more critical with LLMs.

The optimization of the loading time of a model is a multi phase activity that starts with the packaging of the model. This aspect is described in details in Chapter 3 but let's focus now on the steps that are performed from the creation of the deployment in Kubernetes to when the runtime is ready to process requests.

Runtime image provisioning
: The vLLM runtime like any other traditional workload in Kubernetes is wrapped in a container that is pulled from a registry to the Kubernetes node. The image size of vLLM is not small, it is usually few gigabytes (less than 5 GB) and most of the space is usually for the GPU framework like CUDA in case of NVIDIA so it cannot be removed. By default the download of the image from the registry to the node is performed on the fly when the image is required by a Pod. The frequency of this activity is specified with the `imagePullPolicy` property and it is important to avoid `Always` value to refrain that the image is downloaded

for every replica or new deployment. In a production configuration it's usually configured with `IfNotPresent` so that the download happens only the first time or even `Never` making sure that the image is pre-pulled to the node. When the image is already available on the node, the time to load it is very limited so the only aspect to pay attention to optimize this step is to avoid `Always` as `imagePullPolicy`.

Retrieval/Mounting of the model
: The method chosen for storing and retrieving a model significantly impacts loading performance, where a poor design choice can become the primary bottleneck. Common options include downloading on the fly from Hugging Face, storing on S3-compatible storage, copying to a PVC (Persistent Volume Claim) and mounting as a volume, or packaging as an OCI (Open Container Initiative) image. Of these, direct download from Hugging Face and the S3 option are quite inefficient, often taking many minutes due to data transfer and local copying; conversely, PVC and OCI approaches prevent this local copying by mounting the model directly as a volume or sidecar container. If specific requirements necessitate using Hugging Face or S3, it is highly recommended to leverage the KServe Local Model Cache option. This feature configures a local cache using the model's storageUri as a key, ideally combined with fast storage hardware like SSDs (Solid-State Drives) via NVMe (Non-Volatile Memory Express) protocol. Thanks to the KServe Local Model Cache, the performance of Hugging Face and S3 options effectively becomes equivalent to that of using a PVC.

Start of the runtime
: The startup time of vLLM usually takes about a second or less, it produces useful information in the logs as explained in Example 4-1 but from a loading time perspective it is not critical.

Loading of the model
: Regardless the option that has been selected to retrieve/mount the model, when the vLLM runtime starts the model has to be available to be loaded and most of the time is spent loading the weights of the model copying them to the GPU memory. The work to reduce the time to have a new replica of the runtime ready to serve the model is mainly focused on this particular phase. This work has a physical upper bound that is the I/O bandwidth to the GPU memory so it is not possible to get faster than that but the loading time of a model without optimization is pretty far from the physical limit. There are three different extensions in vLLM with projects that are specialized on the loading time problem: CoreWeave's Tensorizer, Run:ai Model Streamer and fastsafetensor. Run:ai Model Streamer is a fast and highly concurrent implementation of the loading procedure that loads the tensors, it supports different file formats including safetensor and different storage options. CoreWeaver's Tensorizer and fastsafetensor both

requires the model to be prepared with a specific serialization format that is then used to make the model faster. In general the configuration and the usage in vLLM is similar for both Tensorizer and Model Streamer, see Example 6-5, while fastsafetensor requires some additional environment variables to be set. Tensorizer and Model Streamer have larger adoption compared to fastsafetensor that is designed specifically to optimize the loading from NVMe devices. Given that Model Streamer doesn't require to repackage the model it is the easiest option to experiment with, but given that this chapter is about production optimization it makes sense to consider the other two options too and pick the one that fits better your specific setup.

Example 6-5. vLLM usage of Run:ai Model Streamer

```
vllm serve \
 --port=8080 \
 --model=/mnt/models \
 --served-model-name=meta-llama/Meta-Llama-3-8B \
 --load-format runai_streamer \                      ❶
# --load-format tensorizer \                         ❷
 --model-loader-extra-config '{"concurrency":16}'    ❸
```

❶ The value `runai_streamer` as value for `load-format` parameter doesn't requires a different serialized format but it enables the Model Streamer code

❷ Using `tensorizer` as value it enables Tensorizer loader but this also requires to have the model saved with Tensorizer serializer

❸ This configuration enables 16 concurrent threads that will load the model in parallel. The other available options can be found in the documentation of the environment variables of Model Streamer

Exposing the model
When the model has been loaded vLLM exposes the OpenAI compatible API and many other endpoints and it is ready to serve requests. The list of APIs includes a `/health` endpoint that can be used to configure the readiness probe as recommended by Kubernetes best practices.

These are the main steps performed by vLLM to start the runtime, load the model and expose it. The two phases that takes most of the time is the download of the model and the loading of the model in the GPU. We explained how to avoid the download time with proper configuration and how to improve the loading time of the model using one of the extensions of vLLM. The only other advice that has been mentioned is to use fast storage options like a NVMe device.

Applying all of these recommendations can reduce time to scale up vLLM from many minutes to tens of second based on the size of the model.

LLM-aware routing

Load balancing the requests across the different replicas is a challenge that starts to impact the cluster as soon as a single replica is not enough and multiple replicas of the model are provisioned. The default strategy that Kubernetes has to dispatch the requests across multiple replica is round robin, the requests are uniformly distributed and it has been defined considering the workload of microservices where monitoring CPU and memory is enough to monitor the current workload of the single replica and decide when to scale up/down.

In reality the round robin approach has already showed limitation when Kubernetes is deployed on cloud environment with multi-zone configurations: Kubernetes introduced the topology aware routing to manage via heuristic the load to keep the traffic within the zone it originated from.

With the serving of LLMs the boundaries of Kubernetes are stretched more than in the past and this requires to explore a different approach to optimize the routing and the load balancing of a request to a replica. There are multiple factors that should be considered to perform specialized routing of the requests.

Each request is different
: This aspect has been already highlighted in the previous chapters, there is no correlation between the size of the input prompt and the number of tokens the model will produce and the generation can continue for many seconds. The impact of a request on a replica is not predictable, thus the routing strategy must consider the actual work the replica is performing and in particular how many requests are still waiting to be processed. There is a specific metric, `vllm:num_requests_waiting`, produced by vLLM to keep track of this information.

Batching
: vLLM creates batch of requests of a certain size that is configurable to use all the available resources and produce more tokens. This is critical in realtime scenario to handle more requests in parallel, but it is not always possible to fill the entire batch with requests and the router can mix offline inference requests with realtime to fill a batch.

Prefill and decode workload
: We already explained the different impact on the hardware that prefill and decode phases have. In particular the prefill phase is directly related with the size of the prompt so it is possible to have dedicated instances of vLLM designed to

perform the prefill of large prompts. This is called disaggregated prefill and it is covered in "Disaggregated Serving" on page 177.

KV Cache reuse

The KV cache has been mentioned multiple times already in the previous chapters and it is a critical aspect to make more efficient the generation of the tokens but this impact doesn't involve only a single request: the model has no memory, every request is considered as something completely new but for example in a chatbot, the whole conversation is provided for every new message including all the previous messages to be processed (prefill phase) again. The same pattern applies with AI Agents when a tool is invoked and the result is sent back to the model together with the previous prompt. A router that is aware of the status of the KV cache of each replica can route the request exploiting this aspect and it is called prefix-aware routing.

Different service level requirements

Realtime requests have higher priority than batch requests but the prioritization of the requests from different users might be more complex. For example the scheduling logic can drop a request that is not critical when the capacity (number of requests waiting and KV Cache size) is less than a certain threshold.

LoRA adapters

There is another use case where the traditional Kubernetes routing pattern doesn't apply well and it is the efficient serving of fine tuned model. The customization of a model is covered in "Tuning a Model" on page 188 but in general from a serving perspective, when a model is fine tuned, the training job produces a new specialized version of it that is deployed as a new, independent model. But when a particular technique, named Low-Rank Adaptation (LoRA) (see "Low-Rank Adaptation (LoRA)" on page 191), is used, the fine tuned model is saved as thin layer, called LoRA adapter, to be composed with the base model enabling the possibility to be deployed in the same runtime instance as adapter saving hardware resources but breaking the mapping of one model per endpoint.

There is a lot of interest to optimize the inference and reduce its cost so there are different initiatives that aim to solve or at least improve both the two different scenarios described before and it is a field that is still evolving. The target goal that essentially all the different initiatives have in common is to obtain a gateway component that is aware of the LLM traffic and that it is able to optimize it. See Figure 6-2 for a high level representation of this component.

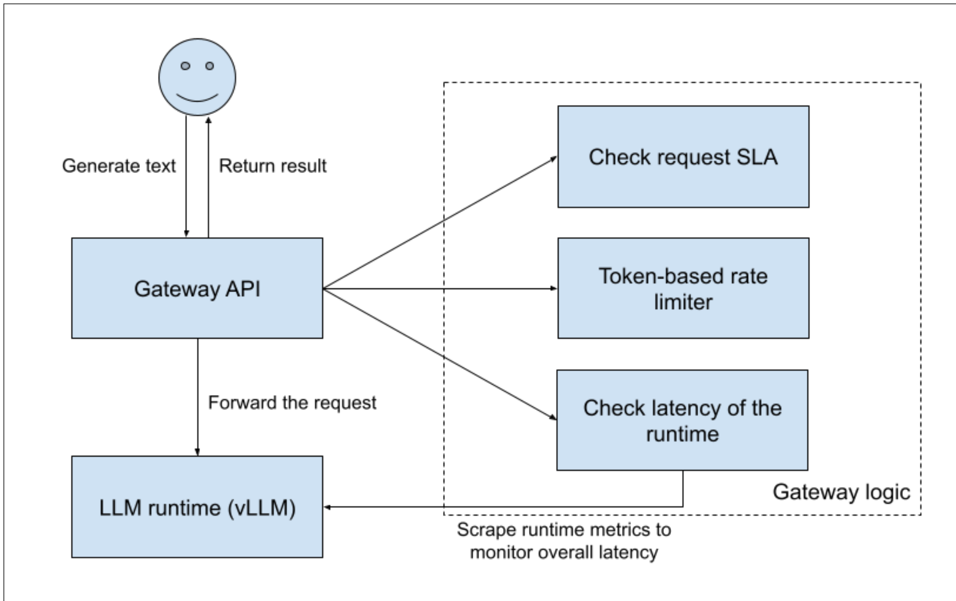


Figure 6-2. LLM Aware Gateway API

Let's start from the last scenario of LoRA adapter that is the simplest from a routing perspective because it is enough to make it aware of the mapping between runtime instance and LoRA adapter. The main challenge is that there is no one-to-one mapping between the endpoint and the model and the routing logic of the cluster should support this service discovery logic to forward the request for a LoRA fine tuned model to where it is available.

The vLLM runtime has native support to serve LoRA adapter together with the base model, making both models available under the same endpoint and giving the user the ability to specify which model to execute directly in the request. See Example 6-6 to learn how to serve LoRA models with vLLM.

Example 6-6. Serving LoRA adapters with vLLM

```
vllm serve meta-llama/Meta-Llama-3-8B \           ❶
    --enable-lora \                               ❷
    --lora-modules my-lora-model=$HOME/.cache/huggingface/  ❸

...
curl localhost:8080/v1/models | jq
{
    "object": "list",
    "data": [
        {
            "id": "meta-llama/Meta-Llama-3-8B",   ❹
```

```
            "object": "model",
            ...
        },
        {
            "id": "my-lora-model",                         ❺
            "object": "model",
            ...
        }
    ]
}

...
curl localhost:8000/v1/completions \
    -H "Content-Type: application/json" \
    -d '{
        "model": "my-lora-adapter",                        ❻
        "prompt": "LoRA is a",
        "max_tokens": 10,
        "temperature": 0
    }' | jq
```

❶ The base model is served with vLLM.

❷ LoRA support should be enabled with a proper parameter.

❸ This parameter lists all the LoRA adapters to load, `my-lora-model` is the name of the model and after that there is the local path, thus inside the container, where the LoRA adapter is saved. It is a list so it is possible to load multiple LoRA adapters and the folder where the model is can be a mounted volume from outside.

❹ The base model is included in the list of available models.

❺ The LoRA model is listed as an additional available model.

❻ It is possible to specify the name of the LoRA model directly during the execution in the same way it is done for base models so there is no differences from an end user perspective.

Now that we know how to use vLLM to serve multiple LoRA adapters, the next step is to make Kubernetes aware of it so that this information can be used during the routing of a request. There are projects in Kubernetes to manage the network and the Kubernetes Gateway API should become the standard way to declare and configure all kinds of gateway. What we need is an AI Gateway!

One of the most active community is the Kubernetes Special Interest Group (SIG) dedicated to model serving, WG-Serving, that incubated the Gateway API Inference Extension project designed to optimize the routing and efficiency of LLM serving.

The project is also sometimes referred to as the Inference Gateway, it extends Kubernetes Gateway API to bring awareness of the Inference workload.

Gateway API

The Gateway API project is a very big and complex project focused on L4 and L7 routing in Kubernetes. It is an official Kubernetes project with the final goal to define the next generation API and implementation of Kubernetes Ingress, Load Balancing and Service Mesh.

It is role-oriented and it defines a different set of APIs to represent all the aspects of network configuration, from the infrastructure provider to expose a single application.

The `Gateway` describes how traffic can be translated to Services within the cluster but it is just the *intent* not the actual endpoint or full specification. The creation of a route is protocol specific like `HTTPRoute`, it is attached to a Gateway and defines the rule to forward a request to a `Service`.

There are many other objects and concepts as part of the full specification that are not critical to explain the Inference Extension so they are not going to be covered here, more detailed information can be find on the API Overview page of Gateway API website.

The first feature that has been developed under the Gateway API Inference Extension project to optimize the routing is a LoRA aware routing API to allow users to declare a *pool* of inferences that represents a single instance of the runtime that it is serving one base model and multiple LoRA models. The two APIs that have been introduced are `InferenceModel` and `InferencePool`. The `InferenceModel` just represents a model and its configuration while the `InferencePool` object represents a set of pods where the runtime is running and the extension that it is used to routes to them. See Example 6-7 for the usage of these APIs.

Example 6-7. Example of Gateway API Inference Extension usage

```yaml
apiVersion: inference.networking.x-k8s.io/v1alpha2
kind: InferenceModel
metadata:
  name: base-model                                    ❶
spec:
  modelName: meta-llama/Meta-Llama-3-8B
```

```yaml
  criticality: Critical                                   ❷
  poolRef:
    name: vllm-llama3-8b                                  ❸
---
apiVersion: inference.networking.x-k8s.io/v1alpha2
kind: InferenceModel
metadata:
  name: my-model
spec:
  modelName: my-model
  criticality: Default
  poolRef:
    name: vllm-llama3-8b                                  ❹
  targetModels:
  - name: my-model-1                                      ❺
---
kind: InferencePool
metadata:
  labels:
  name: vllm-llama3-8b
spec:
  targetPortNumber: 8000
  selector:
    app: vllm-llama3-8b
  extensionRef:
    name: vllm-llama3-8b-epp                              ❻
---
apiVersion: v1
kind: Service                                             ❼
metadata:
  name: vllm-llama3-8b-epp
spec:
  selector:
    app: vllm-llama3-8b-epp
  ports:
  ...
```

❶ The base model is deployed and registered as valid target

❷ It is possible to specify how critical is a model compared to the others

❸ This is the name of the pool that is the single entry point for all the models

❹ It refers to the same pool because it is part of the same deployment

❺ This is the name of the LoRA model

❻ The extension is used to wire the `InferencePool` with the `Gateway`

7. This spec for the `Service` is not full because it is a traditional service that targets the runtime

LoRA aware routing is the first use case that has been implemented but the concept of `InferencePool` is very flexible: it represents a set of Inference-focused Pods and the routing logic can use information from different sources to decide which pod should process the request or even to refuse it based on priorities.

Gateway API Inference Extension has been designed with the idea to extend the routing decision logic to be more and more specific for LLM workload. Envoy proxy is the core component that enables this flexibility, it is a highly scalable HTTP proxy implementation used for many different use cases thanks to the possibility to plug custom processing logic via External Processing. The External Processing (ext_proc) filter defines a gRPC protocol that can be implemented by external services to be registered as a processing step with the ability to read and modify both the HTTP headers and body of the request.

The Gateway API Inference Extension project adopts the External Processing concept to define the Endpoint Picker (EPP) protocol. An Endpoint Picker, as the name suggested, is able to pick an endpoint from the `InferencePool` and each implementation of this component must support Envoy External Processing protocol so that it can be invoked by Envoy proxy during the processing.

One of the most interesting Endpoint Picker implementation is the inference-scheduler that is part of llm-d project (see "Disaggregated Serving" on page 177 for more details). This specific Endpoint Picker implementation supports different filters and scoring logic, for example it is possible to configure the scraping of the metrics from the different vLLM instances and use `vllm:num_requests_waiting` metric to pick the replica where there is the lowest number of requests waiting.

From a Kubernetes perspective each Endpoint Picker is a different deployment that is usually deployed in the same namespace where the model is deployed but it is not required, it is only required that it has access to the container where vLLM is running to collect the metrics and that the instance of the Gateway (Envoy proxy) is able to reach the Endpoint Picker. The communication can be secured via mTLS or in general via certificate.

In "Disaggregated Serving" on page 177 the Inference Gateway will be used together with other components in a more complicated setup to distribute the inference workload but the usage of a smarter routing logic has already a big impact in terms of scalability.

Gateway API Inference Extension is not the only option and neither the only open source project focusing on the creation of an AI Gateway. Envoy AI Gateway is another project using Envoy proxy to build the AI Gateway capabilities reusing

Gateway API Inference Extension but extending it with other user facing features like token-based rate limiting and security. Another example is vLLM Production Stack project that has main innovation has introduced an external and sharable KV cache storage to extend the benefit of reusing the content of KV cache across different instances.

Finally there is the already mentioned llm-d project that extends Gateway API Inference Extension to integrate it more and more with vLLM runtime, includes distributed and shared KV Cache and disaggregated prefill. It is one of the most advanced solution currently available for large scale LLM deployments and it will be used as reference in .

Disaggregated Serving

In addition to LLM-aware routing, there are many other optimizations that can be applied to scale the LLM service in production, the more stringent the latency and scalability requirements, the more complex the configuration becomes.

Disaggregated Serving approach distribute the serving of each LLM integrating an LLM-aware router with distributed KV cache and disaggregated prefill optimizations. Before going more in details to describe these techniques it is important to highlight that similar configuration makes the deployment more similar to an appliance instead of a traditional Kubernetes deployment and it is designed for very large scale deployments where few models are served in a single cluster.

Multiple projects has been created with the goal to implement disaggregated serving, the most famous are NVIDIA Dynamo and llm-d and the main difference is the different focus: NVIDIA Dynamo is specialized and deeply integrated with NVIDIA hardware while llm-d aims to support different hardware and integrate existing open source projects leveraging the ecosystem.

Everything that has been described in this chapter up to this section is applicable to a quite traditional Kubernetes cluster that has at least one node with GPUs but this is not enough to support disaggregated serving. As soon the runtime is distributed and in particular the KV cache is shared across different deployments the network bandwidth available to share the KV cache blocks became critical and a dedicated network configuration is required to obtain the benefit of the distribution. The traditional Pod network interface is usually backed by a Ethernet connection that has up to 10-20 Gbps while the bandwidth requirement is about one order of magnitude higher, about 500-600 Gbps!

NVIDIA developed a specialized network stack to match a similar requirement, it is possible to connect different GPUs using NVLink and NVSwitch to break the limit of Tbps in some configuration.

There are other options that are not specific for NVIDIA hardware and they are based on RDMA (Remote Direct Memory Access) and RoCE (RDMA over Converged Ethernet) to reach up to 800 Gbps. InfiniBand is a famous implementation of RDMA that has been created many years before the Generative AI area to support high-performance computing (HPC) and now similar configuration are not limited to supercomputers but it might become more largely adopted with the evolution of Generative AI workloads.

After the introduction of the additional network requirements to support the distribution of the serving runtime, we are going now to introduce two optimization that can be implemented having a similar cluster available: distributed KV Cache and disaggregated prefill.

Distributed KV Cache

The KV Cache has been mentioned many different times in the book because it is a very critical aspect to make the execution of LLM more efficient so the base intuition behind the distributed KV cache is simple: it should be great if we can store in some external cache the KV blocks to reuse them when necessary. This approach enables two main benefit, the size of the KV cache of an instance is not limited anymore by the available memory and it is possible to share blocks across different replicas. It is easy to imagine that the this optimization has a positive impact only if the time to transfer KV cache data from one instance to another one is very fast, in the range of milliseconds. The idea is quite natural but the implementation is very complex, two new projects has been created specifically with this focus: LMCache and NVIDIA Inference Xfer Library (NIXL). The claim from LMCache is "Redis for LLMs" and implements an API to cache KV blocks and the main benefit applies, as expected, to scenarios where the input prompt is very long (like it usually happens with RAG use case) so that the prefill phase doesn't need to be performed for every new request. On the other hand NIXL project is a small library with a more specific goal: accelerate point to point communication for AI runtimes providing an abstraction over the different type of memory (like GPU and CPU) and storage (from file to remote object store). Both these two projects can be used together and this is what llm-d project does to leverage the benefits of each of them and NIXL in particular is very flexible and it might be used even without disaggregated serving enabling for example the possibility to leverage CPU memory as extension of the GPU memory to have a larger KV Cache. The distribution of the KV Cache has an implication for the LLM-aware router component too because, even if the KV Cache is distributed and accessible by all the replicas, it is way more efficient to forward the request to a replica where the necessary KV Cache blocks are already to avoid a cache miss and a block transfer.

Disaggregated Prefill

Prefill is the first phase of the processing of a request, during this phase the input prompt is processed and the first token is produced. After prefill the decode phase continue to produce token by token until the end of the stream. The first phase is compute bound and it impacts the Time to First Token metric while the second phase is memory bound and impact the Inter Token Latency metric (go back to "Understanding LLM" on page 88 for more details). Given the different nature of the workload, disaggregated prefill split the prefill phase and the decode phase to two different poll of instances so that it is possible to scale and tune the two phases independently: for example if the workload mainly includes long input prompts the prefill phase will need more replicas to process them. The main challenge to enable this approach is that prefill phase is in charge to initialize the KV Cache so it is necessary to transfer it from the prefill instance to the decode instance to continue the generation. Fortunately with the distribution of the KV cache that has been described above can be applied to this use case too. From an implementation perspective disaggregated prefill requires the distributed KV cache and a routing component.

Now that all the ingredients to build a disaggregated serving stack has been introduced, it is possible to describe the end to end architecture of a similar solution. The design described in Figure 6-3 represents llm-d but essentially applies to NVIDIA Dynamo too, there are few differences between the two implementations, probably the main one is that Dynamo supports different inference runtimes while llm-d is integrated with vLLM.

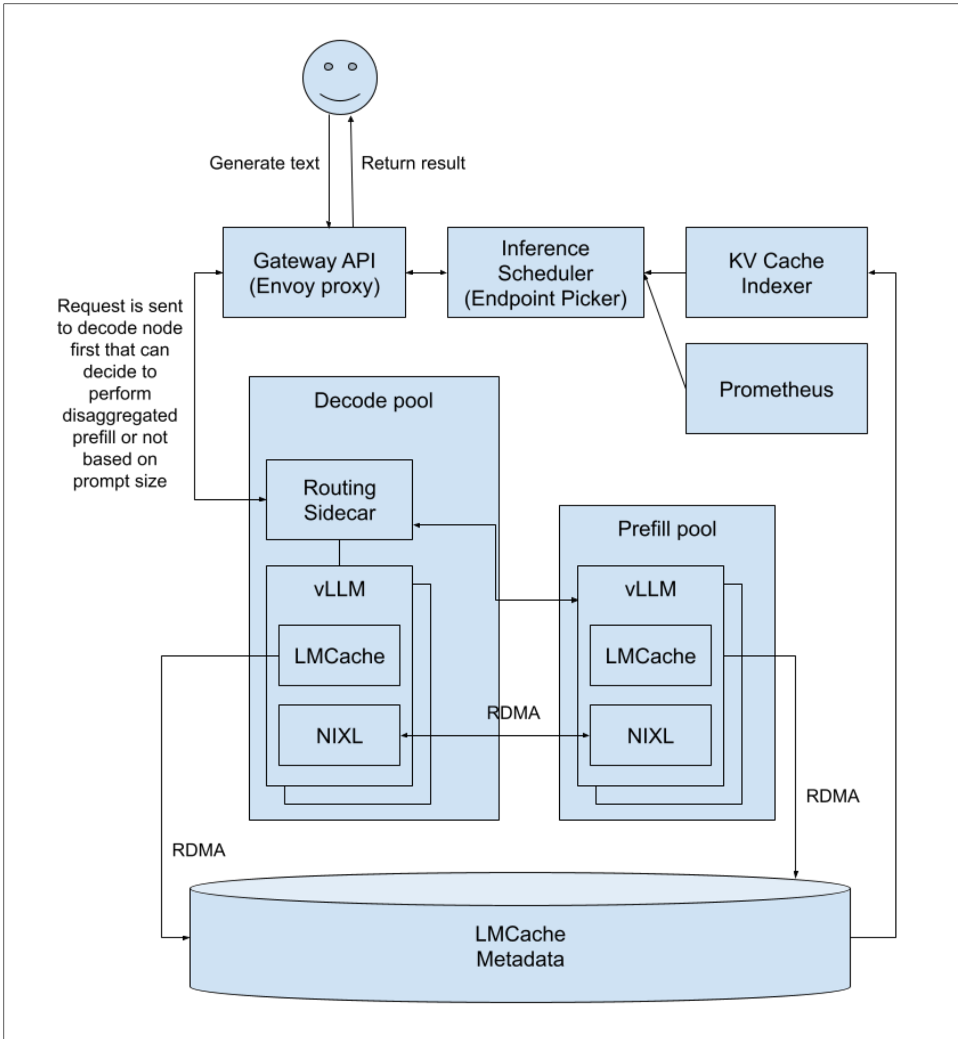


Figure 6-3. llm-d disaggregated serving architecture

The development and optimization of the serving stack is far from completed, new models and new models architecture are defined almost every week and same it happens on the runtime development side. Solutions like disaggregated serving introduces higher coupling of the different components but it is necessary when you have very large scale deployments. The adoption of a similar solution increase the complexity to manage from a platform perspective but there is ongoing work to simplify the deployment and the lifecycle management under llm-d community but also under other projects like KServe. Disaggregated serving is based on the contribution of many different communities, for example the disaggregated prefill

topology has been introduced by Mooncake project and then adopted by NVIDIA Dynamo and llm-d. This is the power of open source development!

Lessons learned

This is the end of one of the most important chapters of the book because it covers the configuration and the tuning of the serving stack, from model tuning and runtime optimization to more complex solutions. The innovation in the LLM space is far from reaching the peak and Kubernetes ecosystem is evolving together to make the execution of LLM more efficient, scalable and easy to manage. One example of this evolution is the LeaderWorkerSet (LWS) project that is introducing a new and better APIs to deploy complex and multi Pod topologies like the one required by disaggregated serving.

Finally, the distributed nature of scaling inference workload raises challenges that makes inference workload similar to distributed training jobs that for example requires topology-aware scheduling. This topic is covered in ???.

There are still many areas of improvement to automate the management but Kubernetes is clearly showing its strengths and flexibility becoming the best platform to operate inference of Generative AI at scale.

CHAPTER 7

Model Customization

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

The creation of a Large Language Model (LLM) from scratch is a complex and expensive multi-step process, affordable for only a few companies in the world. This chapter will not cover creating a model from scratch. Instead, it focuses on tuning an existing LLM to customize its behavior. We will describe several techniques for customizing an LLM and explore the technologies available on Kubernetes to implement and deploy the corresponding training jobs. Before diving into the tuning techniques let's do a step back and briefly describe how a LLM is created and when the tuning phase can happen.

Introduction to LLM creation

The evolution of LLM training techniques is the main arena for competition among the companies creating these models. This race is fueled by billions of dollars in monthly investment to produce new and better models. The internal details of the training procedure are considered critical intellectual property, and the technical

papers published with a model's release rarely cover enough detail for full reproducibility. The technical paper for DeepSeekV3 is a notable exception.

A big part of the innovation centers on new model architectures that includes more efficient ways to compute the attention matrix. However, dataset curation and the specific methods used to tune the model are areas where full disclosure is rare.

The input data mist be prepared to clean up and remove duplication. Then the first phase of the training process, usually called *pre-training*, is where most of the time and cost is spent: all data are processed using thousands of GPUs for many weeks. The result is a base model that is able to predict text but it doesn't know the concept of a task or what appropriate content is.

The next step is *alignment*, the process of encoding human values into the LLM to make it perform tasks as safely and reliably as possible. This phase is analogous to Isaac Asimov's Three Laws of Robotics. Just as each robot has core principles to ensure it is safe for humanity, an LLM must apply human values to solve tasks without being harmful. The alignment phase is more complex overall compared to the pre training. It requires curated labeled data and a reward step (from human and/or dedicated reward models) where every response from the model is classified.

It is possible to find base models that only went through a pre training phase but the vast majority of models that are publicly available have been already aligned so that they are ready to be used for a specific set of tasks. Model customization, also known as *post-training*, applies to an already aligned model. See Figure 7-1 for the high level description of this creation pipeline.

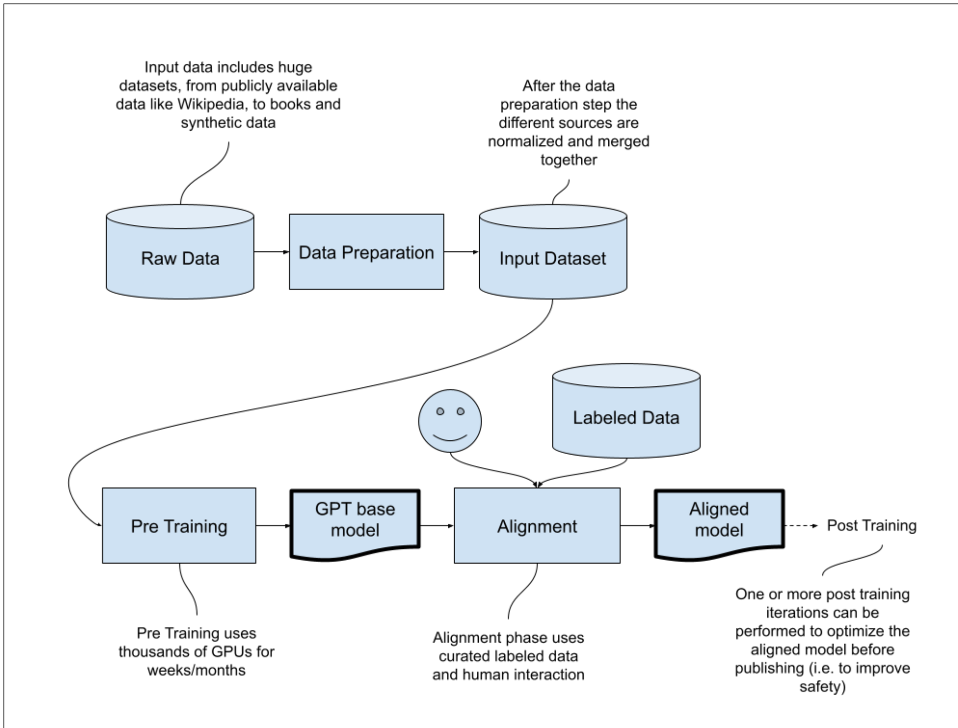


Figure 7-1. LLM creation pipeline

Model Tuning, Model Customization and Post-Training

Model tuning is a general term for various fine-tuning techniques and is not specific to Large Language Models (LLMs), as it also applies to predictive AI.

Model customization is a broader term that encompasses all techniques used to modify an LLM. Some of these methods differ from traditional fine-tuning and may require multiple steps, including human interaction.

Post-training refers to the specific phase in the LLM creation pipeline where model customization occurs. This step can be applied multiple times to incrementally inject new policies or knowledge into the model.

These terms are often used interchangeably in this book because they all involve modifying a model and present similar operational challenges on a Kubernetes platform.

The primary difference that makes generative AI unique from predictive AI is its versatility. A single LLM can perform a large number of different tasks, whereas a

traditional machine learning model is specialized for just one. This versatility is why we covered inference first: you can often adapt an existing LLM for different use cases through prompt engineering alone.

Prompt Engineering

Prompt engineering is the process of crafting detailed and specific instructions (prompts) to guide an LLM's output. This set of instructions are critical to maximize the accuracy of the response. This field is becoming a specialization in its own right, with best practices for communicating effectively with a LLM to obtain the most accurate results.

Effective prompt engineering is not just about specifying the task; it also involves describing:

- The scenario (e.g., "This is an airline company named ABC")
- The role the model should take (e.g., "You are an AI-assistant chatbot to help customers")

The boundaries of the task to limit hallucinations or attacks (e.g., "You can only reply about our company and if you are sure about the answer"). Similar prompts are usually specified by the provider of the service and hidden to the end users as *system prompt*.

Since every LLM is trained on a vast but finite dataset, another use of prompt engineering is to inject additional data into the prompt, forcing the model to use that information during generation.

Basic or manual prompt engineering techniques have evolved into established patterns that make the system more powerful, even enabling it dynamically invoke tools to retrieve information. This is a core principle of AI Agents and is often called *context engineering*. The term reflects that the main engineering work lies in creating the input context for the LLM, a process involving complex, multi-component, and iterative steps.

One of the most widely adopted patterns for context enrichment is Retrieval-Augmented Generation (RAG), which efficiently injects data into the context from external sources filtering out only the content that is related with the user question is included in the context.

With the RAG pattern, additional data is ingested as embedding vectors into a vector database. When a user request arrives, an initial query is performed against the vector database to find content that is semantically *close* to the user's input. This additional context is then included in the prompt for the model to use when answering the question.

This solution helps to overcome the limitation of the context window that every model has: each model has a defined limit in the context window that makes impossible, and neither efficient, to include the entire knowledge base in the input prompt. RAG is a good solution to mitigate the problem because the preliminary filter selects only the data most relevant to the user's question.

See Figure 7-2 for a high level representation of a RAG pipeline.

The embedding concept has been already described in "Understanding LLM" on page 88 and in particular in the section on "Prefill" on page 91.

More of Retrieval Augmented Generation and Context engineering is covered in "Retrieval-Augmented Generation" on page 217 together with AI Agents in "Agentic Workflows" on page 229.

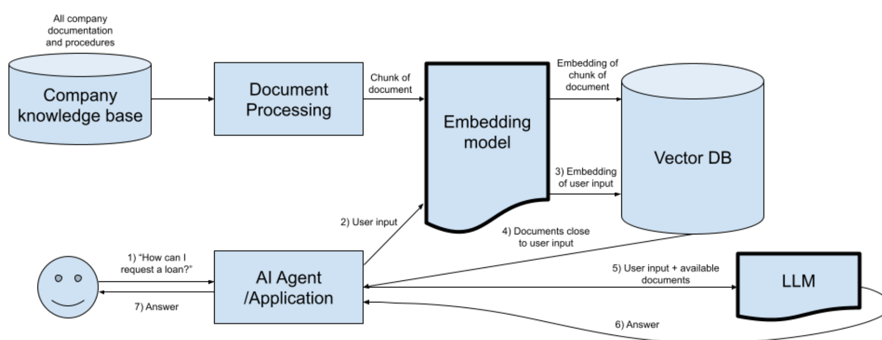


Figure 7-2. An example of RAG pipeline

The possibility to influence model behavior directly in the input prompt is powerful, in particular using the RAG technique, and often enough for many use cases but it has limitations. As described in "vLLM runtime parameters tuning" on page 159, a large context window requires more GPU memory at inference time. The same principle applies to model size: a small, specialized model can be as effective as or even more effective than a larger, untuned model. The flexibility of solutions like RAG where it is enough to update the vector database with new data in a few minutes and *refresh* the knowledge of the solution together with the adoption of patterns of Agentic AI is getting more and more adoption in the market taking over the model customization space. One important aspect to consider is that all prompt and context engineering techniques are completely compatible with model customization so it is possible to apply them to a general purpose model in the same way of a tuned one.

Model customization is a key tool in the generative AI toolbox, particularly for controlling inference costs. It allows a company's core, slow-changing knowledge to

be embedded directly into the model, reducing the need for a large context window with every request.

For example, a bank could create a customized model with embedded domain knowledge about loans, trading, and credit risk. This information, which does not change frequently, would be part of the model itself rather than being provided in the context of every request.

This is just a scenario where model customization can be considered a good fit, especially in conjunction with Small Language Models (SLM) that requires less resources to be served. An SLM usually has between 8 and 16 billion of parameters so it is a good candidate to be tuned with constrained time and resources.

Tuning a Model

The possibility to re-train a model, also known as *post-training*, is not something new to machine learning. In traditional predictive AI, models are often fine-tuned in a second phase to update them with new data. In the context of Generative AI this activity is usually performed to specialize a model and improve the performance in a specific domain and to reduce the overall cost of the solution leveraging a specialized smaller models instead of one of the bigger and more expensive alternatives. The image Figure 7-3 describes the high level process to fine tune a model to embed new knowledge in the original model.

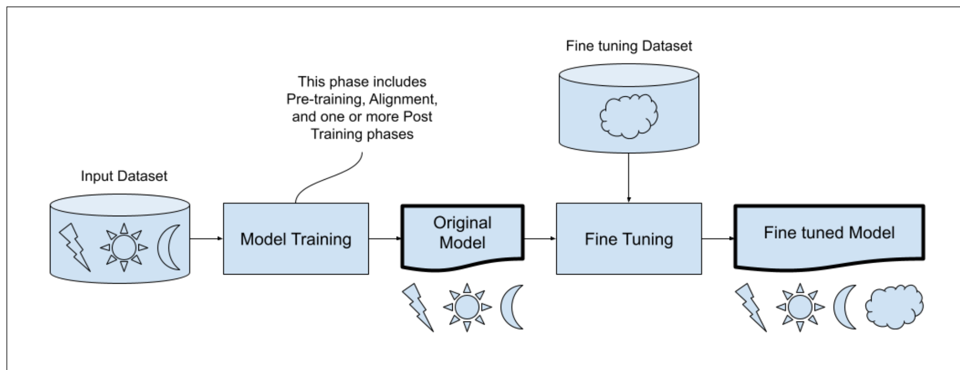


Figure 7-3. Fine tuning concept

While *fine-tuning* is less complex and costly than pre-training by an order of magnitude, it can still take many hours or even days to run. Sometimes, however, a full re-training is unnecessary, for example the user might want to reduce the domain areas that the model should be able to answer similar to the prompt engineering use case described before but as builtin feature in the model and less affected by external attacks. These simpler options fall under a category named *Parameter-Efficient Fine-Tuning (PEFT)*.

Hugging Face is an open-source community, platform, and company that aims to democratize access to machine learning, with a particular focus on natural language processing (NLP). It serves as a central hub for sharing models, datasets, and libraries.

In particular for both full fine-tuning and PEFT approaches, Hugging Face provides a utility library called SFTTrainer that can load a model and perform various tuning techniques, including an evaluation step to compute accuracy.

The name of the library `SFTTrainer` stands for Supervised Fine-Tuning Trainer. The term *supervised* is usually omitted when discussing fine-tuning because the process is implicitly supervised.

While some techniques for unsupervised fine-tuning exist, the vast majority of methods require labeled data as input, data that has been classified by a human or another model. The reason is straightforward: for a model to learn a specific policy or piece of knowledge, the input dataset must contain the specific traits the model is expected to embed.

The creation of a supervised input dataset is usually an expensive activity. As a result, these curated datasets are orders of magnitude smaller than the datasets used for unsupervised pre-training.

Fine tuning

Fine-tuning a model involves continuing the training process to embed additional knowledge. In other words, full fine-tuning changes all the model's parameters, producing a brand-new model that is completely independent of the original. This approach requires a considerable amount of labeled data—at least hundreds of thousands of new examples—to influence the model enough to learn new concepts. It is a very expensive activity. From a Kubernetes platform perspective, it requires many GPUs during the training phase and dedicated GPUs to serve the new model, as there is no efficient way to layer or merge it with the original at inference time. While this is the primary approach for predictive AI, full fine-tuning is far less common in generative AI due to the high cost of preparing the dataset and the overall expense of both training and inference. As mentioned earlier it is possible to use Hugging Face `SFTTrainer` to perform this type of fine tuning (Example 7-1).

Example 7-1. SFTTrainer usage to perform Supervised Fine-Tuning

```
from datasets import load_dataset
from trl import SFTTrainer
from transformers import AutoModelForCausalLM

# Load the dataset with new content for the model to learn.
# This can be a public dataset from Hugging Face or a local file.
```

```
train_dataset = load_dataset("json", data_files="my_file.json")
# The function used to load the model is the same one used for
# inference.
# The model can be downloaded on the fly, but it is typically
# downloaded locally first.
original_model = AutoModelForCausalLM.from_pretrained(...)

trainer = SFTTrainer(
    model=original_model,
    train_dataset=train_dataset,
)

# This method triggers the training phase.
trainer.train()
# Save the new version of the model to a target location.
trainer.save_model("target_location")
```

Parameter-Efficient Fine-Tuning (PEFT)
: PEFT is a group of techniques that takes a different approach to tuning a model. The original model remains unchanged; instead, it is composed with a new layer that influences its behavior at runtime during inference. This can be seen as an advanced form of prompt engineering, where the customizing prompt is embedded in the model. From a Kubernetes platform perspective, PEFT is much easier to manage for both training and serving. The training phase requires fewer data samples (between 100 and 1,000 labeled examples), making the training job shorter and less hardware-intensive. Serving these fine-tuned models is also more efficient because the base model can be dynamically composed with one or more tuned layers at runtime in the same deployment, thanks to support in modern inference engines. We cover efficient model storage in "OCI image for storing model data" on page 74 and inference routing in "LLM-aware routing" on page 170. The main drawback of PEFT is that it has a more limited impact on the model compared to full fine-tuning, which modifies all parameters. With PEFT, only a small fraction of the parameters are affected. For example, Low-Rank Adaptation (LoRA), one of the most popular PEFT algorithms, might tune less than 1% of the total parameters for a Llama 3.1 8B model. Hugging Face created a library named `peft`` to collect different PEFT algorithms, and it integrates natively with the `SFTTrainer` class (Example 7-2).

Example 7-2. LoRA fine tuning using SFTTrainer

```
from datasets import load_dataset
from trl import SFTTrainer
from peft import LoraConfig
from transformers import AutoModelForCausalLM

# The loading of the model and the dataset is equivalent to
# the previous example with full fine tuning.
```

```
train_dataset = load_dataset("json", data_files="my_file.json")
original_model = AutoModelForCausalLM.from_pretrained(...)
lora_config = LoraConfig(...)                                      ❶

# To enable PEFT, you just need to pass the lora_config instance
# as the peft_config argument.
trainer = SFTTrainer(
    model=original_model,
    train_dataset=train_dataset,
    peft_config=lora_config,                                        ❷

)

trainer.train()
trainer.save_model("target_location")
```

❶ The only difference is the initialization of the configuration for the PEFT technique being used, in this case, LoRA. There are many parameters, check Hugging Face peft documentation for more details.

❷ To enable PEFT, you just need to pass the `lora_config` instance as the `peft_config` argument.

Low-Rank Adaptation (LoRA)

LoRA is one of the most widely used PEFT techniques. Instead of fine-tuning the full model, LoRA freezes the original model and trains a small number of new parameters on the new data.

Instead of fine-tuning the full model, LoRA keeps the original model frozen and trains new, smaller matrices of parameters called adapters. These low-rank matrices learn the updates, and their product is combined with the original weights.

In a traditional fine-tuning job, the training process learns a new, full-sized matrix representing the weight updates. LoRA, however, decomposes this large update. Instead of learning the full matrix, the training produces two much smaller, low-rank matrices. When these two smaller matrices are multiplied, their product approximates the full weight update. This decomposition is what makes the training procedure significantly more efficient. See Figure 7-4 for a graphical representation of this process.

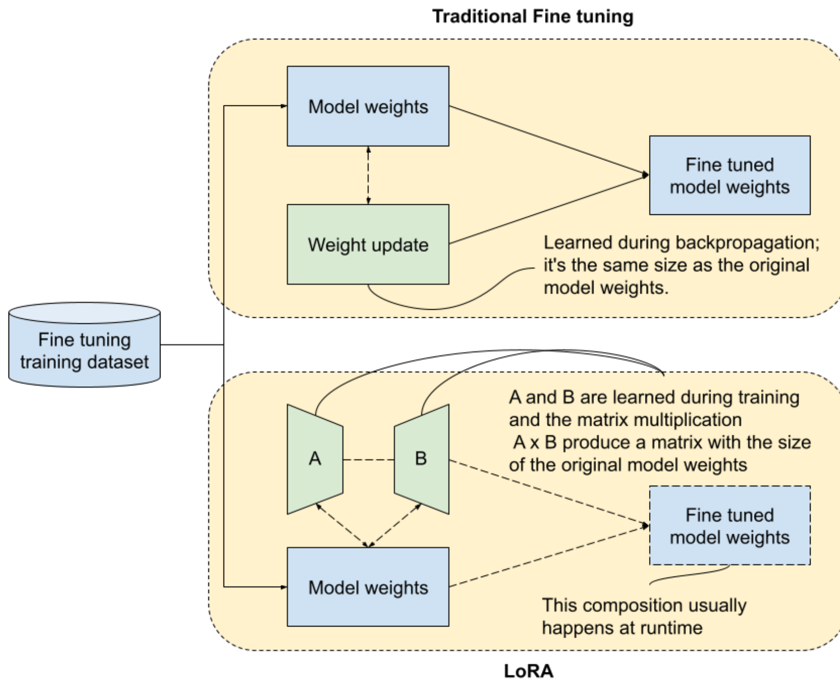


Figure 7-4. Comparison of LoRA decomposition and full fine tuning

LoRA is applicable to a large set of LLMs, and many variants of the algorithm exist for specific scenarios. Two notable specializations are X-LoRA, which extends the approach to Mixture-of-Experts (MoE) architectures, and QLoRA, which applies quantization to reduce fine-tuning memory requirements.

LoRA offers two main benefits: a cheaper training phase (in terms of time and hardware) compared to full fine-tuning, and an efficient inference approach. Since the base model is not modified, adapters can be composed with it at runtime. This makes it possible to serve one base model and many LoRA-tuned models using the hardware required for only the base model.

See Figure 7-5 for a visual representation of LoRA adapter serving.

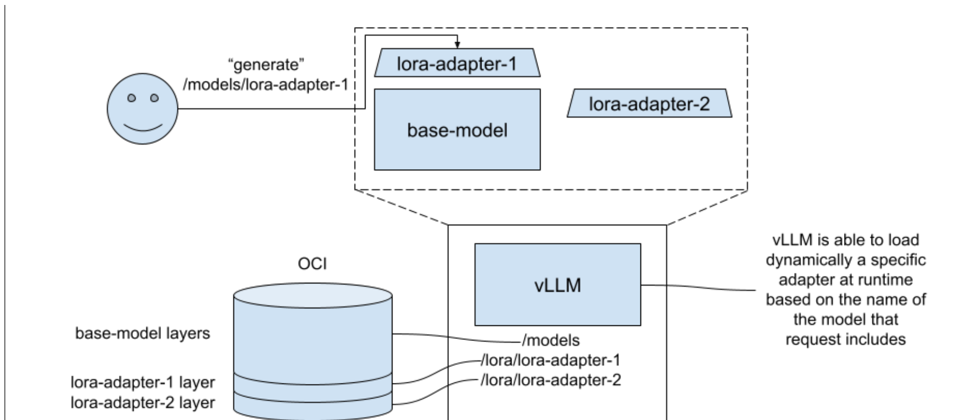


Figure 7-5. Serving of LoRA adapters

Even if it is not the traditional use case for LoRA, it is still possible to merge the LoRA adapter with the base model for testing purposes. See Example 7-7 which explains how to implement this.

We suggest the blogpost Practical Tips for Finetuning LLMs Using LoRA (Low-Rank Adaptation) from Sebastian Raschka to learn more about LoRA.

Advanced Tuning techniques
: Fine-tuning and PEFT are not the only ways to tune a model; new and more complex techniques are constantly emerging. Many of these new approaches involve multi-step workflows rather than a single training loop, which can include using synthetic data produced by the model in a previous iteration. Some of the most common advanced techniques include: Group Relative Policy Optimization (GRPO), Direct Preference Optimization (DPO), InstructLAB, Model Distillation, Model Merging, and Reward Modeling.

This book will not cover these advanced methods in detail, as each technique is a complex topic. They are also very different; for example, GRPO is an innovation from the DeepSeek team, while InstructLAB is a full methodology from IBM Research for *alignment tuning*. To learn more, we recommend the Transformer Reinforcement Learning (TRL) library from Hugging Face, which collects many of these techniques with dedicated trainer classes.

The focus of this book is on the operational challenges of generative AI. From a Kubernetes platform perspective, these tuning methods manifest as long-running, multi-deployment topologies where most components require dedicated GPUs and the ability to communicate securely. The security of this communication is critical for production workloads and is covered in ???.

Running Tuning Jobs on Kubernetes

So far, we have introduced the core concepts for creating and tuning an LLM, from traditional full fine-tuning to PEFT and advanced tuning pipelines. Understanding these different approaches is important because they have different implications and challenges from a Kubernetes platform perspective.

Now, let's shift from the implementation details to the platform requirements. All these tuning techniques have at least one training phase that requires GPUs for scaling. The GPU management principles covered in previous chapters for inference largely apply here as well. Refer to Chapter 5 for a recap of how to configure Kubernetes for GPUs and schedule workloads that require them.

Although provisioning GPU workloads is not new, a major additional challenge for training is that networking can easily become the bottleneck of the system. A tuning job is not equivalent to an inference request; even for an SLM, the hardware requirements for tuning are greater than for serving. As a result, the job will likely require multiple GPUs on the same node or even across multiple nodes (Figure 7-6).

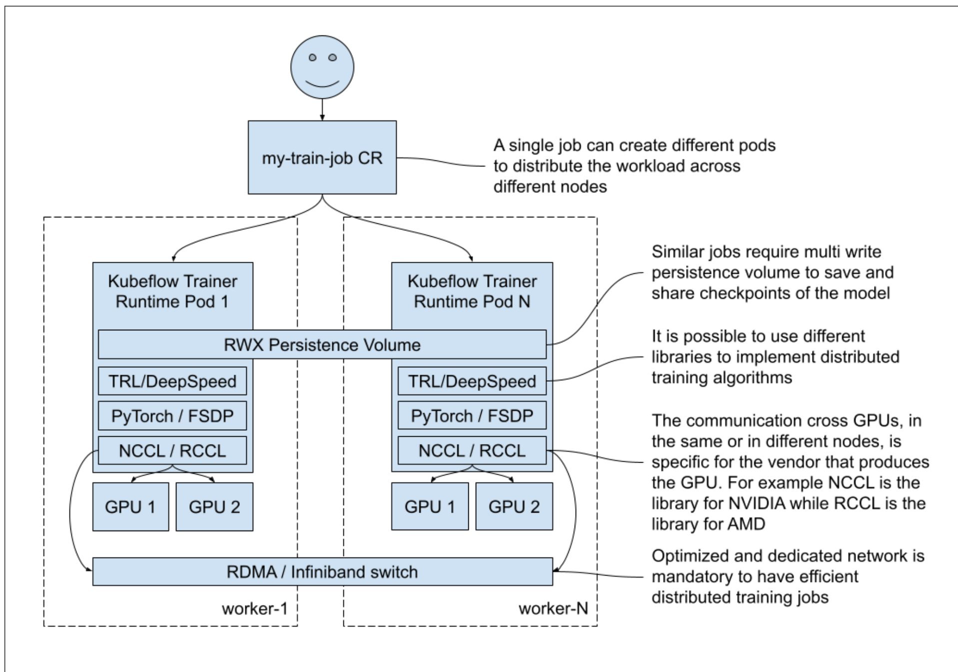


Figure 7-6. Multi-node training job

In a similar scenario, based on the type of tuning performed, the sharded weights of the model are gathered on all GPUs before every "step" of the execution of the model (in particular every layer forward and backward passes). This action requires

a continuous stream of data shuffling across the GPUs and, based on the size of the model and the number of the GPUs, it can produce traffic of many Gigabytes per second. The bandwidth is the main scalability challenge and requires improvements across the entire stack, from specialized network interfaces and protocols to more efficient kernel implementations and ad-hoc GPU instructions. Similar to inference optimization, training also has kernel implementations that benefit from dedicated GPU instructions, such as the Liger Kernel (optimized for Triton) and FlashAttention.

The attention kernel is a core component, and it is usually embedded in a higher-level, end-user library. While Hugging Face provides many of these libraries, other options include DeepSpeed and NVIDIA's Megatron-LM.

Although these libraries have different APIs and configurations, they all use PyTorch, which has become the de facto standard deep learning library for LLM implementation.

PyTorch is an open-source machine learning library originally created by Meta and now owned by PyToch Foundation that is part of the Linux Foundation.

It has many different applications but in the context of LLM development is mainly used as core deep learning library: the other end user libraries like Hugging Face `transformers` uses PyTorch and deprecated the support for other deep learning libraries like TensorFlow or JAX.

PyTorch project has many different packages that covers a large set of capabilities, from the core neural network implementation to a compiler and to a distributed package with the specific goal to support distributed training jobs. In particular Fully Sharded Data Parallel (FSDP2) is the most common library used to scale the job on multiple nodes.

The software and hardware stack is evolving rapidly, with the hope that many of these complexities will eventually become implementation details from a platform perspective. However, optimizing the network stack is a challenge that cannot be avoided and is covered in ???.

Fortunately the entire Kubernetes ecosystem is evolving with the goal to make the platform better and better at managing Generative AI workloads and in particular Kubeflow Trainer is a project specialized on the management of fine tuning jobs.

Kubeflow Trainer

Kubeflow Trainer is the component of the Kubeflow ecosystem designed specifically for managing the scaling and distribution of LLM fine-tuning. The Kubeflow project aims to be *the foundation for AI platforms on Kubernetes*, and it is evolving from its origins in predictive AI to support generative AI workloads. We previously introduced another component, the Kubeflow Model Registry ("Kubeflow Model Registry" on page 66), in Chapter 3.

Kubeflow Trainer's sole purpose is to manage the Kubernetes building blocks required to configure, deploy, and scale long-running training jobs. Its API is designed for two different personas: the platform administrator, who configures the cluster and available resources via a TrainingRuntime, and the data scientist/AI engineer, who submits the training job using a TrainJob. Since these roles have different skills and tools, Kubeflow Trainer provides a Python Kubeflow SDK that abstracts the creation of the TrainJob, so the data scientist does not need to interact directly with Kubernetes resources.

Figure 7-7 illustrates the full architecture of Kubeflow Trainer.

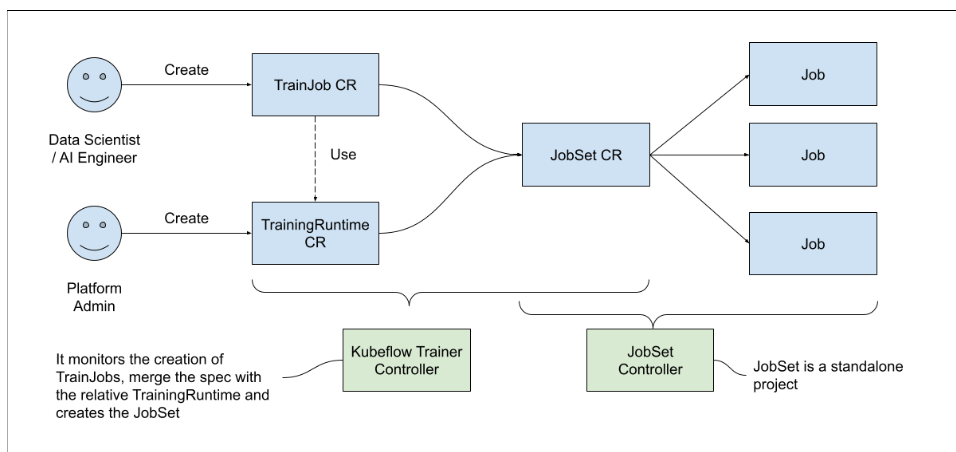


Figure 7-7. Kubeflow Trainer architecture

The `TrainingRuntime` (or `ClusterTrainingRuntime` for cluster-wide configuration) is equivalent to KServe's `ServingRuntime`, which was described for inference in "KServe" on page 38. It's a template that declares the availability of a runtime, such as PyTorch, including its container image and other options. Similar to a `ServingRuntime`, a `TrainingRuntime` is only visible in the namespace where it's created, and `TrainJobs` must be in the same namespace to use it. A `ClusterTrainingRuntime`, however, is visible to the entire cluster.

Kubeflow Trainer is designed to support multiple frameworks for distributed training, such as PyTorch, DeepSpeed, and MLX. Because of this multi-framework design, a TrainingRuntime requires a mandatory `trainer.kubeflow.org/framework` label. The SDK uses this label to apply the correct configuration for the specified framework (e.g., `torch` for PyTorch) and its trainer.

The trainer represents the library that uses the framework to define and perform the training job, it can be a `BuiltinTrainer` or a `CustomTrainer`.

A `BuiltinTrainer`, like TorchTune, provides a pre-defined training script for common use cases like LLM fine-tuning, requiring only parameters for the input dataset and LoRA configuration. While less flexible, it's easier to start with. On the other hand, a `CustomTrainer` gives the user full control by allowing them to define a Python function containing the entire training process. This approach gives the data scientist maximum flexibility, while the administrator only needs to define the `TrainingRuntime` with the approved framework.

The `TrainJob` object includes the source code of the job and references a training runtime, as mentioned before it is not designed to be manually written by the data scientist because the provided SDK simplifies the configuration.

Once the `TrainJob` is created, the Kubeflow Trainer controller merges it with the `TrainingRuntime` to produce a `JobSet` and the corresponding Kubernetes `Jobs`.

A `JobSet` is another resource designed to represent a group of Kubernetes Job, it is a standalone JobSet project that aims unify the API to deploy High Performance Computing (HPC) and AI/ML training workloads on Kubernetes.

The installation procedure of Kubeflow Trainer is straightforward like for any other Kubernetes controller (Example 7-3).

Example 7-3. Installing Kubeflow Trainer

```
export VERSION=REPLACE_WITH_VERSION                                    ❶
kubectl apply --server-side -k \
  "https://github.com/kubeflow/trainer.git/manifests/overlays/manager?ref=${VERSION}"


kubectl apply --server-side -k \                                       ❷
  "https://github.com/kubeflow/trainer.git/manifests/overlays/runtimes?ref=${VERSION}"
```

❶ Replace the value with the version to install, i.e. `v2.0.0`.

❷ While the Kubeflow Trainer project provides a default set of ClusterTrainingRuntimes to simplify the getting started experience, it is expected that administrators will define their own curated list of runtimes for production use.

Kubeflow Trainer provides a set of default `ClusterTrainingRuntime` but they are optional, you skip this specific installation step and replace the default runtimes with one or more custom runtimes (Example 7-4).

Example 7-4. ClusterTrainingRuntime

```
apiVersion: trainer.kubeflow.org/v1alpha1
kind: ClusterTrainingRuntime                                          ❶
metadata:
  name: my-torch-distributed-runtime                                  ❷
  labels:
    trainer.kubeflow.org/framework: torch                             ❸
spec:
  mlPolicy:
    numNodes: 1                                                       ❹
    torch:
      numProcPerNode: auto
  template:
    spec:
      replicatedJobs:
        - name: node
          template:
            metadata:
              labels:
                trainer.kubeflow.org/trainjob-ancestor-step: trainer
            spec:
              template:
                spec:
                  containers:
                    - name: node
                      image: pytorch/pytorch:2.7.1-cuda12.8-cudnn9-runtime  ❺
```

❶ Replace with `TrainingRuntime` to create a namespace-scoped training runtime.

❷ The data scientist uses this name to select the desired runtime for their job.

❸ This label is used by the SDK to guide the configuration of the `TrainJob`.

❹ The spec can define default values for most of the value, for example this means that the job can only use 1 node.

❺ The administrator might want to control the image that is used in the cluster by replacing this value with a customized image. The image is GPU specific so in this case it is for NVIDIA CUDA.

With the cluster configured and the `TrainingRuntime` available, the platform administrator's work is done. The data scientist can now focus on creating the training job (Example 7-5).

Example 7-5. Trainer function using Hugging Face TRL to be used as CustomTrainer

```
def my_custom_trainer(args):
    from datasets import load_dataset                                          ❶
    from transformers import AutoTokenizer, set_seed
    from trl import SFTTrainer

    # It is not mandatory to set a fixed seed but it is useful for reproducibility
    set_seed(args["seed"])

    # Load tokenizer
    tokenizer = AutoTokenizer.from_pretrained(                                 ❷
        ...,    # args[...]
        use_fast=True
    )

    # Load Datasets
    train_dataset = load_dataset(                                              ❸
        ...,    # args[...]
    )

    # Initialize Trainer
    trainer = SFTTrainer(                                                      ❹
        model=...,
        args=...,
        train_dataset=train_dataset,
        eval_dataset=...,
        peft_config=...,
        processing_class=tokenizer,
    )

    trainer.train()                                                            ❺

    trainer.save_model(
        ...,    # args[...]
    )
```

❶ The custom trainer function must be self contained so the import must be part of the body. This example is based on Hugging Face libraries: `datasets` for the training dataset (and optionally the evaluation dataset), `transformers` for the tokenizer and `trl` for the actual trainer class. Note that there is no Kubeflow Trainer specific code here, the function is a plain Python train function that can be directly invoked.

❷ The tokenizer is related with the model that is fine tuned. It is important to use a fast tokenizer that works concurrently to avoid the slow down of the training process.

❸ The dataset contains the new knowledge that the model should learn during the fine tuning, it can be a publicly available dataset but most likely it is going to be a custom one.

❹ The initialization of the `SFTTrainer` is equivalent to the previous example. This is where the model is selected, the datasets are specified, and the PEFT technique (e.g., `LoraConfig`) is configured.

❺ The `train()` method initializes the training process. Hardware configurations, such as the number of GPUs and workers, are not specified here; instead, they are defined when the job is created (see Example 7-6).

With the training logic and configuration defined in the trainer function, you can now create the TrainJob using the Kubeflow Python SDK.

Example 7-6. Create TrainJob via Kubeflow SDK

```
from kubeflow.trainer import CustomTrainer, TrainerClient                    ❶

client = TrainerClient()

torch_runtime = client.get_runtime("my-torch-distributed-runtime")

job_name = client.train(
    trainer=CustomTrainer(
        # The custom trainer function is injected here with its parameters
        func=my_custom_trainer,
        func_args=...,      # load_args()
        num_nodes=8,                                                         ❷
        resources_per_node={
            "cpu": 4,
            "memory": "64Gi",
            "nvidia.com/gpu": 1,
        },
    ),
    runtime=torch_runtime,
)

client.wait_for_job_status(name=job_name, status={"Running"})                ❸
_ = client.get_job_logs(job_name, follow=True)

# It is possible to get all the steps and the status for each of them
# steps = client.get_job(name=job_name).steps
```

```
# client.delete_job(job_name)  ❹
```

❶ In this example a CustomTrainer is created using the custom function defined in the previous example and the TrainerClient is used to submit the `TrainJob`.

❷ Hardware requirements are specified during the submission of the job. The hardware requirements are directly related with the size of the model and the type of tuning that is performed. In this example the value has been used to customize a Meta-Llama-3.1-8B-Instruct model using peft LoRA.

❸ The client can wait for a specific job status. This is a blocking call; in the example, it is used to wait until the job is running. You can also fetch logs or configure the use of TensorBoard. TensorBoard is a visualization toolkit, originally from the TensorFlow project, that is now compatible with multiple libraries, including PyTorch.

❹ While you can delete the job at any time, even while it's running, this action also removes the `TrainJob` object. This is not recommended, as it erases the record of the job run.

A training job like the one in Example 7-6 requires many parameters, definitely more than 10, and if the job creation code runs inside a Jupyter Notebook (maybe using Kubeflow Notebooks component), it is possible to easily configure all the parameters using `yamlmagic` library.

This project is a Python module that can be installed in a notebook via `pip install yamlmagic`, loaded via `%load_ext yamlmagic` and after that it is possible to initialize a variable, like `my_params`, using a code block that begins with `%%yaml my_params`. Each row of the block after this first line is parsed as a YAML and `my_params` and becomes a Python dictionary ready to be used.

In the LoRA example, the training procedure doesn't produce a new, full model. Instead, each saved checkpoint is a LoRA adapter that can be dynamically composed with the base model at runtime. This enables the efficient serving of multiple tuned models, as described in Figure 7-5. While this is the best approach for efficient serving, it can be useful for testing purposes to merge the LoRA adapter with the base model to create a new, standalone model. This scenario is covered in Example 7-7.

Example 7-7. Merge LoRA adapter with the base model

```
from peft import PeftModel
from transformers import AutoModelForCausalLM

base_model = AutoModelForCausalLM.from_pretrained(                          ❶
    ...,
    device_map="cuda"
)
finetuned_path = "/opt/app-root/Meta-Llama-3.1-8B-Instruct/checkpoint-100/"  ❷

model = PeftModel.from_pretrained(base_model, finetuned_path)
merged_model = model.merge_and_unload()                                      ❸
merged_model.save(...)
```

❶ The base model must be loaded first, the `device_map` parameter makes the model directly load on GPU.

❷ It is necessary to have the path where the LoRA tuned model is stored, after every training epoch a new checkpoint (aka model candidate) is created and in this example the checkpoint number 100 is selected.

❸ After the base model and the fine tuned layer are loaded together, it is possible to merge and obtain the new model using `merge_and_unload()` method.

This example covers the usage of the Kubeflow Trainer project from both the administrator's and the data scientist's perspectives, showing how they can collaborate on a distributed model customization project. However, this is just the first step.

From a platform perspective, scheduling these long-running, resource-intensive workloads requires additional optimization to ensure fair cluster usage and prevent the under-utilization of hardware, particularly GPUs. One significant challenge not covered here is gang scheduling. Distributed workloads often require all their pods to be deployed simultaneously to run correctly—an "all-or-nothing" semantic. ??? focuses entirely on these platform optimizations, including a dedicated section on gang scheduling.

The experience for the data scientist is simpler, as the Kubeflow ecosystem allows them to focus on the model customization lifecycle with limited awareness of the underlying Kubernetes platform.

The Kubeflow project includes numerous components to support the entire MLOps/LLMOps lifecycle. The Kubeflow model registry was covered in "Kubeflow Model Registry" on page 66, with a focus on model metadata management, while Kubeflow Trainer is covered in this chapter to enable distributed training jobs.

The development and management of the Python code included in the fine tuning example is something that data scientists can create by leveraging two other Kubeflow components: Kubeflow Notebooks and Kubeflow Pipelines.

Kubeflow Notebooks manages the infrastructure for web-based IDEs like Jupyter, making it easy for data scientists to self-provision an environment and experiment with the Kubeflow Trainer SDK.

After experimenting and defining the training job, a data scientist can use Kubeflow Pipelines to convert the notebook into a reproducible pipeline. This allows the logic to be executed multiple times for retraining the model, either by extracting the code into distinct steps or by directly incorporating the notebook into the pipeline.

Other frameworks

The Kubeflow Trainer project takes a Kubernetes-native approach to managing the lifecycle of distributed training jobs, allowing both platform administrators and data scientists to work with their preferred tools.

While Kubeflow Trainer and Hugging Face's TRL offer a robust, platform-centric solution for distributed training on Kubernetes, several other projects and libraries like DeepSpeed, Ray, and Unsloth provide specialized tools to optimize the fine-tuning process, particularly focusing on efficiency, speed, and resource management for Large Language Models (LLMs) not only on Kubernetes.

DeepSpeed
: DeepSpeed is a deep learning optimization library that wraps PyTorch to simplify the management of training jobs. Using DeepSpeed with Kubeflow Trainer is very similar to the previous example. You only need to select a DeepSpeed-compatible `TrainingRuntime` (such as the default `deepspeed-distributed` runtime) and update the custom trainer logic (Example 7-8).

Example 7-8. Trainer function using DeepSpeed

```
def my_custom_deepspeed_trainer(args):
    from transformers import AutoModelForCausalLM, AutoTokenizer, set_seed
    from datasets import load_dataset
    from torch.utils.data import DataLoader
    from torch.utils.data.distributed import DistributedSampler
    import deepspeed
```

```
# Initialize DeepSpeed distributed training
deepspeed.init_distributed(dist_backend="nccl")                    ❶
local_rank = int(arg["local_rank"]])

# Set seed for reproducibility
set_seed(args["seed"])

# Load tokenizer
tokenizer = AutoTokenizer.from_pretrained(..., use_fast=True)   # args[...]

# Load Datasets
train_dataset = load_dataset(...).with_format("torch")      # args[...]     ❷

train_loader = DataLoader(
    dataset, batch_size=16, sampler=DistributedSampler(dataset)
)

# DeepSpeed configuration
ds_config = {
    ...,    # args[...]
}

# Initialize DeepSpeed engine.
model_engine, _, _, _ = deepspeed.initialize(                      ❸
    model=model,
    config=ds_config,
    model_parameters=model.parameters(),
)

num_epoch = int(...)    # args[...]                                 ❹
for epoch in range(num_epoch):
    for batch_idx, batch in enumerate(train_loader):
        for key in batch.keys():
            batch[key] = batch[key].to(local_rank)
        outputs = model_engine(batch)
        loss = outputs.loss

        model_engine.backward(loss)
        model_engine.step()


model_engine.module.save_pretrained(...)   # args[...]
tokenizer.save_pretrained(...)      # args[...]
```

❶ You must initialize the distributed training. The value `nccl` is used for NVIDIA CUDA hardware, and `local_rank` is an environment variable provided as an argument to the training script.

❷ The example loads the dataset using the Hugging Face datasets library, and it can be easily converted to a PyTorch dataset.

❸ The engine initialization returns multiple variables, but for this example, only the `model_engine` is needed.

❹ In this example, the training loop is explicit, showing the computation of the forward pass, the loss, and the backward pass.

Ray

While Kubeflow Trainer's flexibility is sufficient for most model customization techniques, it is not the only framework for distributed computation on Kubernetes; the Ray project is a valid alternative. Ray provides an entire ecosystem of components for AI platforms and was previously introduced in sec-deploying-models-ray-serve-kuberay>>. Its core concepts, like the `RayCluster` (Figure 2-5), are generic and apply to the training space as well. Ray's integration with Kubernetes is managed by KubeRay, which provides the necessary APIs. This allows you to deploy a `RayCluster` and then submit a R`ayJob to perform a long-running, multi-node computation for model customization. The process is similar to the Kubeflow Trainer example: you create a Python script with the training logic, instantiate a `RayCluster` (a step not required by Kubeflow Trainer), and then deploy the job. A full example of using DeepSpeed and Ray to fine-tune an LLM can be found in this repository, which uses the CodeFlare SDK to programmatically configure KubeRay resources.

It is important not to confuse Ray Tune with LLM model tuning. Ray Tune is a module designed for hyperparameter tuning and optimization, which mainly applies to predictive AI.

The equivalent project in the Kubeflow community is Kubeflow Katib.

While not designed for model customization, it is still possible to use Ray Tune with the Hugging Face transformers library for hyperparameter optimization techniques like Population Based Training (PBT) as described in this example.

Unsloth

The Unshloth project specifically targets LLM customization of LLM with the goal to make it easy, fast and with limited hardware requirements. It has a large and active community. While not designed for large-scale infrastructure on Kubernetes, it is very easy to start with, as it can be installed locally as a standard Python package (`pip install unsloth`). In this respect, it can be seen as the fine-tuning equivalent of local inference projects like Ollama or llama.cpp. Although designed as a local library, it is possible to deploy it on Kubernetes using the AIKit project.

Lessons learned

After focusing on the inference stack, this chapter introduced the basics of LLM creation and the different approaches to customizing a model.

The model customization ecosystem is large and evolving rapidly, but key players are already emerging. PyTorch has become the standard deep learning library, while libraries like TRL and DeepSpeed are the primary sources for trainer implementations. As new techniques are developed, they are likely to be incorporated into these projects.

From a Kubernetes management perspective, the Kubeflow Trainer project provides a flexible and robust infrastructure to simplify the management of fine-tuning jobs. It offers a well-defined API that recognizes the distinct roles of the data scientist and the platform administrator.

We are now ready to focus on hardening the model customization infrastructure for multi-user, large-scale, and enterprise Kubernetes clusters in the next chapter.

CHAPTER 8

AI-Driven Applications

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

In previous chapters, we demonstrated how to deploy model servers like vLLM on Kubernetes, package model data, and operate inference at scale. Building on that foundation, we will now shift from serving single models to architecting complete AI-driven applications where a LLM is just one of many component.

This chapter focuses on application architecture: how requests flow through a system, how context is retrieved or tools are invoked, and how state is maintained over time. We will introduce popular architectural patterns, the key components of AI application stacks, and the challenges of integrating LLMs into real-world applications. To maintain a clear focus on the architectural overview, we will keep the discussion at a high level. We will dive deeper into more concrete technical developments in the next chapter.

LLMs started their march of conquest into mainstream software as chatbots, with ChatGPT as its most prominent representative. Chat is still the dominant interaction pattern, but the software behind chat has grown up. Modern AI apps wrap an LLM

with application logic that fetches business context, calls internal systems, and writes state. The LLM inference service is a powerful component, but it does not reach into databases or call tools by itself[1].

The application is in charge and uses the LLM for generation or reasoning. You will see where to use retrieval for grounding, when to orchestrate tool calls, and how to keep state across turns without losing control of cost, latency, and quality.

In the next section "Architectural Patterns" on page 208 we will see two fundamental setups for embedding such AI-driven applications in a wider operational landscape.

Following that architectural overview, we'll shed some light on important concepts for creating AI-driven applications, namely *Retrieval-Augmened Generation* in "Retrieval-Augmented Generation" on page 217 and *Agentic workflows* in "Agentic Workflows" on page 229.

By the end of this chapter, you'll have a good understanding of the categories of AI-driven application and how generative AI workloads integrate into broader systems.

Let's jump now into the general architecture and deployment topologies for AI-driven applications on Kubernetes.

Architectural Patterns

Before we dive into the typical architectures of AI apps, let's recap the most important Kubernetes workload types so we can map them to the architectural components we describe.

Mapping each responsibility to the right Kubernetes primitive allows decoupled lifecycles and release cadences. For example, the LLM serving instances might be updated on a different schedule than the application logic deployment. This separation lets you upgrade or scale one part without disrupting others and aligns well with microservice best practices now applied to LLM-centric apps.

Kubernetes Workload Types

Let's have a closer look at the key Kubernetes primitives and their AI app roles. Each of those types are described in more detail in Kubernetes Patterns. We reference the correspondings patterns below in italics.

1 As of late 2025 there has been a growing tendency to hand over more functional responsibilities to the inference platforms. For example, vLLM started to incorporate the Responses API, which includes tool and MCP server calling, a domain previously reserved to dedicated orchestrating middleware like Llama Stack

Deployment
: Used for stateless services that are always running, for example the main application backend or an event-driven orchestrator. Deployments manage rolling updates, scaling, and restart for these long-running components. In an AI app, the orchestrator handles requests or events, while the LLM inference server, often with Graphics Processing Units (GPU) requests, handles inference. Both run as Deployments, which allows them to scale independently of each other. See the *Declarative Deployment* pattern.

StatefulSet
: Used for stateful services that need stable identities or persistent storage. Examples include databases, caches, or vector stores that keep embeddings and context. StatefulSets ensure these components survive restarts with their data intact. See the *Stateful Service* pattern.

Job/CronJob
: Used for one-off or scheduled tasks such as offline ingestion, report generation, or periodic maintenance. CronJobs trigger Jobs on a schedule, while Jobs run to completion and free resources after. See the *Batch Job* and *Periodic Job* patterns.

Ingress/Gateway
: Provides entry into the cluster for client requests. A standard Kubernetes Ingress Gateway routes external requests to the appropriate service. For AI-aware routing and scheduling strategies, refer to "LLM-aware routing" in Chapter 6. The underlying techniques are described in the *Service Discovery* pattern.

By assigning the right Kubernetes primitive to each function, you set lifecycle boundaries in the system. Stateless logic in Deployments can be updated and scaled independently, stateful data stores in StatefulSets maintain continuity, and ephemeral or scheduled work happens in Jobs that incur cost only when needed. These boundaries also hint at different Service Level Agreements (SLOs) and cost profiles, for example a conversational API may need low latency and high availability, while a nightly summarization Job can run with relaxed timing.

Keeping these workload types in mind will help you map the architectural components we describe onto a Kubernetes setup.

Let's start with the most popular category of AI applications: UI-facing chat applications like ChatGPT.

Chat Applications

The first example in Figure 8-1 is a chat-facing application. A user talks to a web or mobile UI, which calls a conversational backend. That backend orchestrates the flow: it retrieves relevant context from stores, calls domain tools or APIs when needed, builds a prompt, and then calls the LLM service for generation. After receiving the

model output, the backend post-processes the result, updates per-user memory or other state, and returns the response to the client. This split keeps the LLM focused on generation while the app owns data access, side effects, and policy enforcement.

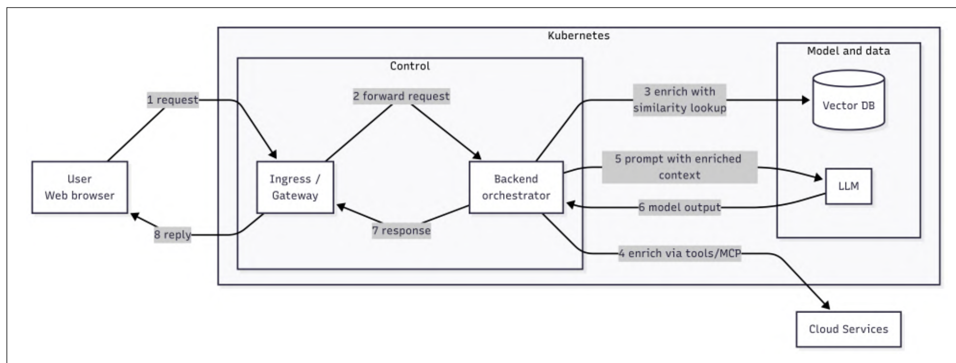


Figure 8-1. A typical chat-like application

This request-response driven system exemplified by a chat application exhibits a linear flow of interactions. A user issues a request and waits for a response. The flow is synchronous from the user's perspective, even if multiple steps occur behind the scenes.

In such a chat-oriented AI app, the LLM is one component in the request flow, while the backend service is responsible for the overall conversation logic.

A typical sequence for a single user query is as follows. For the various components we add the proper Kubernetes workload type like in (*Deployment*).

1. **Gateway or Ingress** routes the user's request to the backend service (*Ingress*, *Gateway*).
2. The **AI Orchestrator** receives the request and controls the chat logic. It retrieves relevant context, invokes tools or external APIs, assembles the prompt, and calls the LLM service for a completion. It retrieves relevant context, invokes any tools or external APIs as needed, assembles the final prompt for the LLM, and calls the LLM service to get a completion. This component implements the "brain" of the chat application - e.g. using frameworks like LangChain, within its code to manage the prompt, memory, and tool usage. In "Retrieval-Augmented Generation" on page 217 and "Agentic Workflows" on page 229 we will focus on exactly this orcherstrator component and how it is designed to deliver its value (*Deployment*).
3. **LLM Service** performs inference and returns the model output. The orchestrator calls this internal service (or potentially an external API like OpenAI) with the prepared prompt and awaits the result. By isolating the LLM in its own

service, we can scale or update the model independently of the application logic. Kubernetes admins might deploy this with GPU nodes and use auto-scaling to handle variable load. Chapter 2 covered how to deploy and scale model servers; here we see them integrated into an app (*Deployment*).

4. **State Management** provides conversation memory or retrieval indexes in a database, cache, or vector store. This state can be stored in a database or cache, such as a Redis or Postgres instance for chat history, or a vector store for embeddings. On Kubernetes, these stateful components are typically run as StatefulSets with PersistentVolumes. The orchestrator service reads previous messages or stored vectors to include in the context ("retrieval"), and after the LLM responds, it might save the latest user query and LLM answer to this store. In our Kubernetes mapping, this database or vector index is a long-lived component that you scale or backup as needed with replication if required for high availability (*StatefulSet*).
5. **Response to Client** may include post-processing or final tool calls before sending the reply.

This pattern keeps the LLM focused on generation or reasoning, while the application retains control over data access, tool use, and side effects. The entire chain runs in a single synchronous request cycle, meaning low latency is a priority. To meet SLO demands (say, sub-second or a few seconds per response), the Kubernetes setup would keep the orchestrator and LLM pods running and ready. Techniques like autoscaling might be employed to handle bursts, but you wouldn't spin these up from zero for each request due to startup time.

A benefit of having a user facing application architecture is that it is easily possible to participate in distributed authentication workflows like OAuth2 that involve browser redirects to authentication servers. This makes security setups simpler and more straight forward than when dealing with backend services described in "Backend AI Services" on page 212.

This architecture separates concerns so that the conversational logic (often updated frequently as prompts and tool integrations evolve) is decoupled from the LLM model serving (which might only change when a new model or version is available). The database can be treated as an external dependency that changes rarely. This decoupling means each piece can be upgraded independently - for instance, deploying a new version of the orchestrator Deployment with improved prompt handling, without touching the LLM Deployment or wiping any stored data. It also allows scaling each component based on its usage: e.g. many chat sessions mainly load the LLM and database, so scale those up, whereas the orchestrator code might be lightweight compute-bound and need fewer replicas.

Backend AI Services

The second pattern is a more interconnected microservices architecture where an LLM-powered service operates as part of a broader system without a direct user interface or direct request to an LLM service. Instead, the LLM logic is triggered by events or calls from other services in the platform.

Figure 8-2 represents this pattern with an AI-driven backend service performing an order risk analysis in an e-commerce platform where multiple services and stores are orchestrated together. In this example, an application orchestrator receives order events from services like *Orders*, *Payments*, and *Catalog*. It then calls an AI risk analyzer that uses a LLM, a vector store of policy text, and past cases to evaluate the order. The analyzer may also call domain tools, such as a rules engine or fraud APIs. Finally, the orchestrator writes the decision to a risk database and emits events for downstream services like *Fulfillment*. This backend can be implemented synchronously with service calls or asynchronously on an event bus.

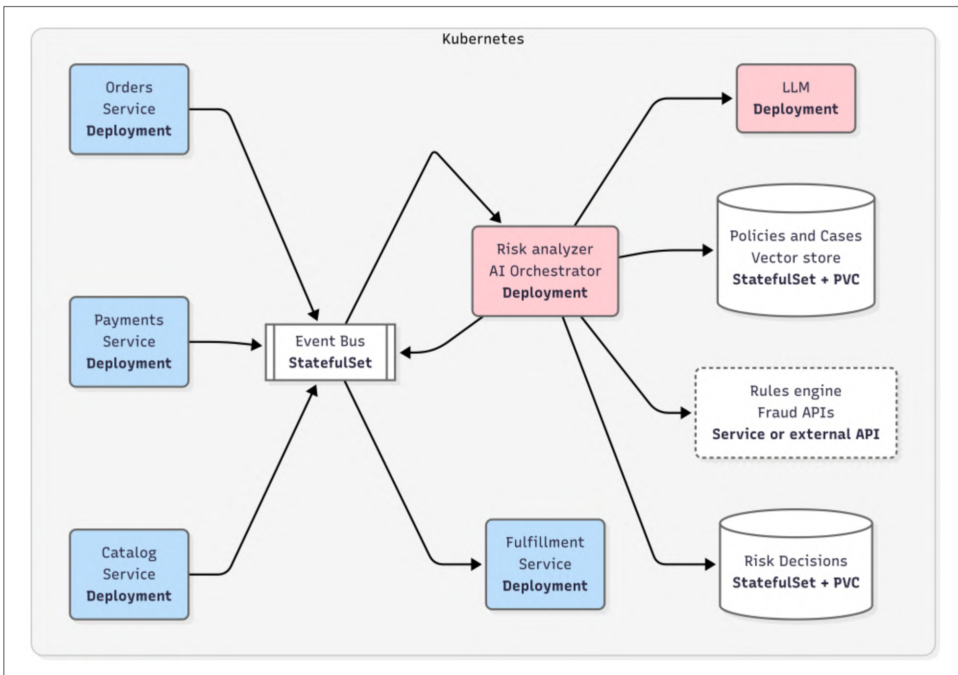


Figure 8-2. Event-Driven AI Service

In this event-driven architecture, the AI application subscribes to events, performs multi-step analysis with an LLM, and emits results for other services.

1. **Message Broker** receives events like `OrderPlaced` or `PaymentProcessed` from business services. Our AI orchestrator service subscribes to the relevant events. In Kubernetes, this might be done via an event streaming platform running in the cluster, or an integration with an external bus. While Kubernetes doesn't natively provide a message queue, many cloud-native systems run on K8s for this - or one might use Knative Eventing or Dapr pub-sub components. The key is that the AI service is triggered asynchronously rather than via HTTP request (*StatefulSet*).

2. **AI Orchestrator** wakes on relevant events and reacts accordingly. For instance, on an `OrderPlaced` event, this service might gather data from multiple sources: fetch the order details, payment history, and relevant product info, then call an AI Risk Analyzer component to evaluate fraud risk. The risk analyzer itself could involve an LLM call with context. In the example, the service uses an LLM plus a vector store of policy documents and past fraud cases to assess the order. It may also call non-AI tools - e.g. a rules engine or third-party fraud detection API - as part of the analysis plan. This orchestration can be thought of as an agent workflow (LLM deciding actions and calling tools) but encapsulated within a single microservice (*Deployment*).

3. **LLM and Tools Services** run as separate Deployments or external APIs. Similar to the chat pattern, the LLM is often a separate Deployment (or an external API) that the orchestrator calls for the generative or reasoning step. The vector database for retrieval (e.g. storing fraud cases or policies) is a StatefulSet with a PersistentVolumeClaim (PVC), acting as a knowledge base to ground the LLM's decision. Any domain-specific tools (like the rules engine) could be separate services, accessible via their own APIs. In some cases, a tool might even be implemented by triggering a Kubernetes Job - for example, if you have a compute-intensive data processing step that can run asynchronously, the orchestrator might create a Job for it and later collect the result. However, more commonly these tools are just HTTP/gRPC calls to other microservices. The key difference from the linear pattern described in "Chat Applications" on page 209 is that these calls are not directly user-initiated but part of a backend flow (*Deployment*).

4. **State Outputs** persist decisions to a datastore and emit new events for downstream services. In the risk analysis example, the outcome (approve/flag the order, a risk score, etc.) is written to a risk database (another StatefulSet for persistent state) and an event like `OrderFlagged` might be produced for downstream services (Fulfillment, Notifications) to act on. This turns the AI decision into part of the event-driven architecture of the whole system.

5. **Downstream Services** react accordingly, e.g. by halting fulfillment or triggering reviews. From a Kubernetes perspective, those downstream consumers are just other Deployments or Jobs that handle events.

This architecture follows a "short think-act-observe loop" pattern. The AI service receives an input event, uses the LLM to plan and possibly take actions, updates state, and then waits for the next input. It is effectively an autonomous agent within the microservice ecosystem, but it operates within defined guardrails and produces auditable results. The AI orchestrator Deployment could scale out horizontally if the event load is high, though coordination might be needed if events have to be processed in order. One thing to note for this pattern is *idempotency* and *reliability*. Since it's event-driven, you often want the AI service to handle duplicate events or failures gracefully. Kubernetes Jobs can be useful for retryable tasks here, but if our AI orchestrator Deployment crashes or needs to update, a message queue can buffer events until a new pod is ready - adding resilience. This pattern can trade a bit more latency (events introduce slight delays and eventual consistency) in exchange for looser coupling and better throughput scaling. It also can be more cost-efficient: the AI orchestrator isn't doing work unless events arrive, and you can even scale it down to zero replicas with frameworks like KEDA or Knative.

From a release perspective, this pattern usually involves many moving parts that can be updated independently, much like in the chat-application pattern: The orchestrator code might be updated as business logic or policies change. The LLM serving stack might change when a new model or more optimized serving solution is adopted (for example, moving from one model to a larger one, or switching to a distributed serving approach for scale). Data stores and message brokers have their own upgrade paths. Designing clear interfaces (event schemas, API contracts) between these components is crucial so that you can upgrade one service without breaking the whole pipeline - which again echoes traditional microservice best practices, now applied to LLM-centric functionality.

This AI backend microservice architecture also has several variations that are not tied to an immediate external stimulus like with this event-driven approach.

Those headless services also can run asynchronously on its own or being part of larger background workflows. These variation include scheduled jobs, long-running agent loops, or on-demand batch tasks.

Scheduled Batch Jobs

To kick off ingestion, nightly summaries, or periodic fine-tuning use CronJobs which update vector stores or derived artifacts. Figure 8-3 shows a simple setup that uses a `CronJob` to fire up an ingestion job. The Job could run a batch script that loads data, invokes the LLM (perhaps calling a model API or running a local smaller model), writes results (e.g. to a file or database), then exits. Because no user is waiting, you might schedule these for off-peak hours or lower-priority nodes to save cost. The "Document Ingestion" phase of a Retrieval-Augmented Generation pipeline that we describe in "Document Ingestion" on page 221 is a good example: you could have a CronJob that periodically processes new docu-

ments, generates embeddings, and updates the vector store (rather than running that in the request path). This improves efficiency and keeps the user-facing parts fast (*Job / CronJob*). For scheduled or triggered batch jobs, Kubernetes CronJobs and Jobs are the natural workload primitives. They provide failure retries, logs of each run, and isolation of resources per run. For example, you might allocate a larger memory or GPU for a nightly job without keeping that allocation all day.

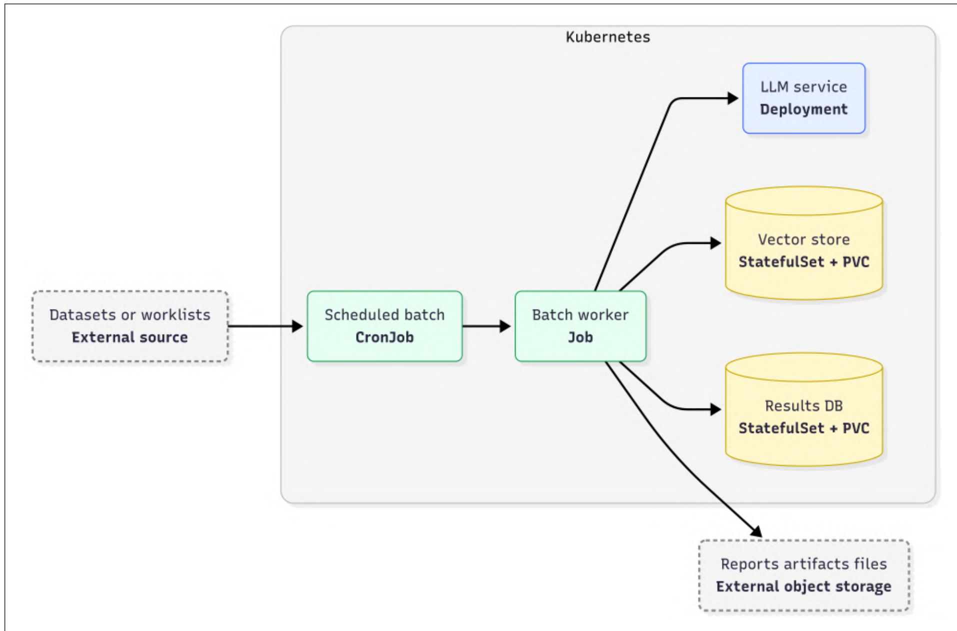


Figure 8-3. Scheduled Batch Jobs

Continuous Control Loops

An alernative to time based trigger, control loops run continuously, watching for certain conditions or iteratively working on a task. Figure 8-4 shows an example of such a polling asynchronous setup. For instance, consider an ambient agent as described in "Ambient Agents" on page 241 that monitors a data stream (logs, social media, IoT sensor readings) and whenever it notices an anomaly or a keyword, it uses an LLM to analyze and perhaps trigger an alert. This agent might run as a Deployment, maybe even with a single replica, that essentially loops: check input, if something of interest, call LLM or tools, produce an output, repeat. Unlike the original event-driven microservice use case layed out in Figure 8-2, this agent may pull for work in a loop (polling a source or awaiting callbacks) rather than react to pushed events. It's conceptually similar to how a Kubernetes controller works (continuous reconciliation loop), except here the "controller" might have an LLM in the decision process. Such an agent could also be user-facing in a passive way - for example, a Slack bot that is always connected

and replies whenever a user mentions it. It's not a request/response on a web API, but a persistent connection with intermittent triggers. For always-on agents, a Deployment as Kubernetes workload type is suitable. You may only need one replica, but you still get benefits like auto-restart on failure. If the agent should not run more than one copy, you could incorporate leader-election logic or use a *Singleton* pattern. One simple way is to run it as a Deployment with replica count 1 and ensure no autoscaling. Another approach is to use a StatefulSet of size 1, though its primary benefit, a stable network identity is often not required for such agents. You find more strategies for singletons in *Kubernetes Patterns*.

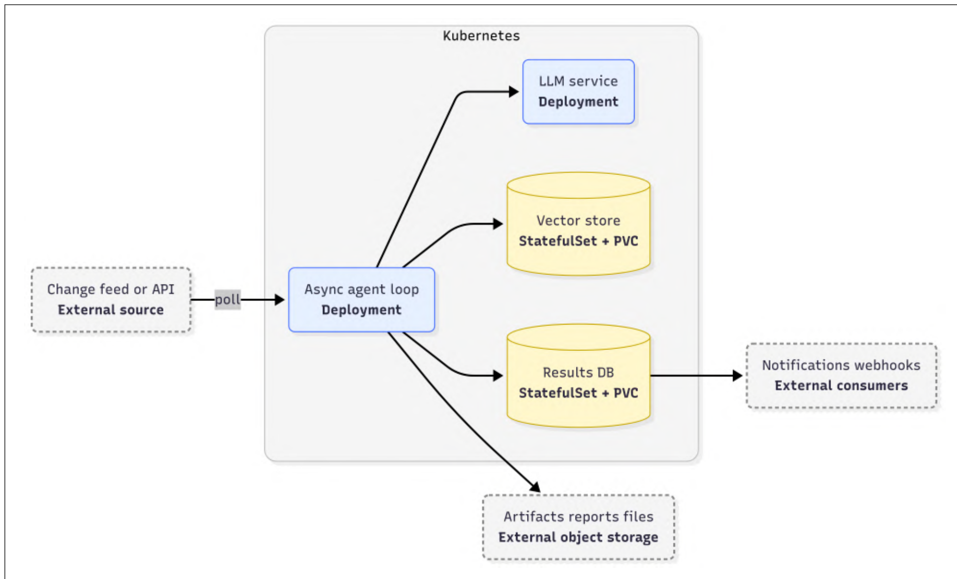


Figure 8-4. Asynchronous Agents

Multi-step Tool Automation

An asynchronous workflow can be also used to accomplish a complex multi-step goal without human intervention. For example, an agent might be tasked to generate and send a weekly summary email. Upon trigger, it will plan steps: query a database, ask an LLM to summarize key points, maybe generate a graph via a plotting tool, and then send the email via an Simple Mail Transport Protocol (SMTP) service. This could be implemented as a Job that encapsulates all steps (with an internal loop for the agent planning), or as a temporary Deployment that runs for the duration of the task (though a Job is simpler for run-to-completion logic). The ReAct pattern and multi-agent coordination discussed in "Agentic Workflows" on page 229 fall here - they are application-level control flows within you orchestration service; they do not map to distinct Kubernetes resource kinds, you'll run them within the Deployment or Job workloas. Archi-

tecturally you must decide whether each multi-step process runs synchronously (holding open a user request) or asynchronously (off the request path). Often, it's safer to run long multi-step agents asynchronously and then notify the user or system when done.

Asynchronous patterns allow you to be more flexible with resource usage. If something isn't urgent, you can run it at lower priority or when capacity is available. For example, if using spot instances or spare cycles, you might schedule non-critical LLM jobs there. Conversely, if an ambient agent is critical (say, watching for security intrusions), you treat it like any important service: ensure it's highly available and fast enough, which might mean dedicating a pod that keeps an LLM model loaded in memory. Always-on agents incur constant cost since the pod runs continuously, whereas event-triggered jobs incur cost only per use - a classic trade-off between cost efficiency and responsiveness. We need to set those lifecycle boundaries intentionally: Which processes can be spun up on demand (to save money) versus which must be pre-warmed and waiting (to meet latency targets).

Now that we covered popular architectural choices for designing AI-infused applications, let's focus on the AI orchestrator component that is central to any AI application architecture. For that we will revisit the popular concepts starting with a technique for grounding LLMs in your domain data.

Retrieval-Augmented Generation

Retrieval Augmented Generation (RAG) is a design pattern that grounds an LLM's output in external data by fetching relevant information at inference time and including it in the prompt. Instead of relying solely on the model's fixed training data, we give the model an "open book" during question answering. The result is fewer hallucinations and answers that reflect the latest, domain-specific knowledge, even when the base model's training data is stale.

It helps to contrast RAG with fine-tuning that we covered in Chapter 7. Fine-tuning teaches new information to a model by updating its weights and is ideal for style, tone, or stable domain patterns you want embedded in the model. However, fine-tuning is resource-intensive and slow, and you must repeat it whenever you have new data. RAG sidesteps retraining by injecting knowledge at query time. You update a vector database with new documents, and the next user query can immediately retrieve and use that information. This makes RAG flexible for dynamic knowledge bases or rapidly changing content and is a big reason it is popular in enterprises. A RAG and model tuning are complementary rather than mutually exclusive. If you have core knowledge that rarely changes, a smaller fine-tuned model can bake in those basics and reduce prompt size. Meanwhile, RAG supplies current or user-specific data that falls outside the model's built-in knowledge. As seen in Chapter 4, large context windows have memory and latency costs, so minimizing prompt size

is beneficial. In practice, many teams combine both: bake long-lived knowledge into a tuned model and use RAG for dynamic or user-specific facts. By offloading static knowledge into the model via tuning and pulling in only relevant facts via RAG, you balance accuracy and efficiency.

The key is that RAG works with any base model, whether original or fine-tuned, because it operates purely through the prompt interface. All prompt-based techniques such as RAG or tool usage are compatible with a fine-tuned model as the LLM backend.

RAG has two distinct phases as illustrated in Figure 8-5. Ingestion prepares knowledge for retrieval.

- **Document ingestion** prepares domain documents by parsing, chunking, embedding, and storing them in a vector database, described in "Document Ingestion" on page 221.
- **User query processing** embeds the user's prompt, retrieves similar chunks, optionally reranks, and assembles the final prompt to the LLM, described in "User Query Processing" on page 223.

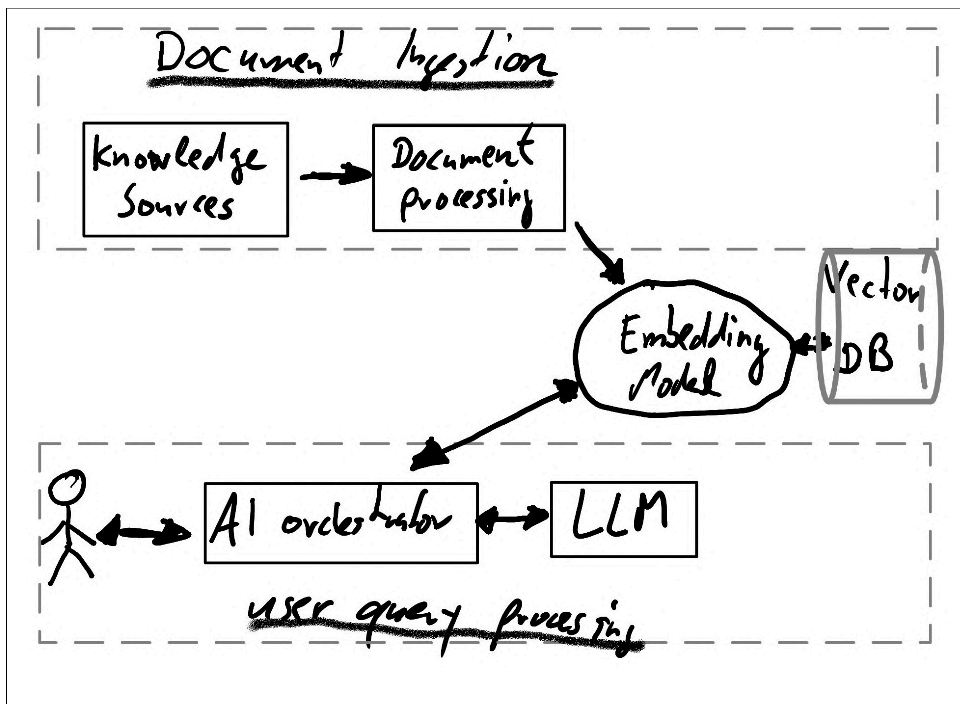


Figure 8-5. The two RAG phases: Document ingestion and user query processing

A RAG setup consists of several cooperating components. Before we learn how those components are operated on Kubernetes, let's look at their responsibilities.

RAG Components

A typical RAG architecture comprises distinct services that map well to microservice boundaries. Figure 7-2 and Figure 8-5 showed an overview of RAG pipelines and their core building blocks. Here we look closer at those components individually, then in "RAG on Kubernetes" on page 226 we map them to the proper Kubernetes workload types.

In this section we discuss the core building blocks of a RAG system indivually. Later in "RAG on Kubernetes" on page 226 we will map these components to Kubernetes workload types and go in on more operation details.

But first let's see what components are included in a RAG setup:

Vector database
: A specialized data store for high-dimensional vectors, also called *embeddings*. The vector database holds your knowledge base in vector form, enabling fast nearest-neighbor search. It returns the documents or snippets most similar to a given query vector. "Vector databases in a nutshell" on page 220 has some more details about and pointers to vector databases. In a RAG system, the vector DB is the "memory" that we query for relevant context.

Embedding model
: The model that converts text or other modalities into embedding vectors during ingestion and at query time. During document ingestion, the embedding model transforms each document or document chunk into a numerical vector, which is then stored in the vector DB. At query time, the same embedding model converts the user's query into a vector so we can search for similar documents by searching for other embedding vectors that are close to the query vector.. The quality of these embeddings directly affects retrieval relevance. Quality here means that semantically similar documents maps to vectors that are close to each other in the high-dimensonal vectorspace. You might use an open-source sentence transformer, a proprietary API (e.g. OpenAI embeddings), or even the LLM's own embedding capabilities if available. The crucial point is to use the *same embedding model* for both indexing and querying. Consistency is key: if you update or change the embedding model, you will likely need to re-embed your documents to maintain search accuracy.

Ranker
: An optional component that improves the relevance of retrieved results. A ranker typically is a second-stage model or heuristic that takes the initial set of results from the vector search and orders or filters them by how useful they are likely

to be for answering the query. For example, a simple approach might rank by similarity score or document metadata like recency or source trust level. More advanced setups use a cross-encoder model or even the LLM itself to score each candidate snippet in the context of the question. Incorporating a ranker can boost answer quality by ensuring only the most pertinent pieces of information get into the final prompt. The trade-off is extra complexity and latency, so whether to use a re-ranker depends on your application's requirements.

Orchestrator

The orchestrator is the glue of the RAG system. As we saw in "Architectural Patterns" on page 208, this central role is common to all AI-driven applications. It handles the overall query workflow. When a user's request comes in, the orchestrator is responsible for calling the embedding model to embed the query, performing the vector DB similarity search, optionally invoking the ranker to refine results, constructing the augmented prompt with retrieved text, calling the LLM service to get an answer, and post-processing and returning the result. The orchestrator could be a custom REST API service you write, an API layer such as Llama Stack or a local AI framework that manages chains of calls like LangChain. The orchestrator often also implements any application-specific rules or guardrails, that, for instance, handles cases where no relevant documents are found, or enforces that certain data sources must be included.

LLM service

Finally, the large language model itself is the component that generates the answer for the end user. The LLM takes the prompt assembled by the orchestrator (which includes the user's question plus retrieved context) and produces a completion. In a RAG setup, the LLM'-s job is constrained to generation. It doesn't need to have all knowledge internally; instead it relies on the provided context for facts. This service could be a model deployment running in your cluster like we have described in Chapter 2, or a call to an external API like OpenAI. The LLM service should be treated like any other dependent service, you send it a request and get back a response, and the orchestrator then delivers that response to the user, often after some formatting or verification.

Vector databases in a nutshell

A vector database (also called a vector store) specializes in fast similarity search so a RAG pipeline can fetch the document chunks most similar to a user's query and pass them to the LLM. Documents and queries are mapped by an embedding model to a single vector in high-dimensional space where semantic neighbors lie close together, and similarity is typically scored with cosine similarity.

Cosine similarity measures how much two vectors point in the same direction rather than how long they are. In two dimensions you can picture two arrows from the

origin forming an angle theta, and cosine similarity is the cosine of that angle: 1.0 when they point the same way, 0.0 when they are perpendicular, and -1.0 when they point in opposite directions. This orientation focus is useful for text embeddings because scaling a vector does not change its meaning, while direction preserves semantics.

Hybrid search combines dense vector matching with lexical ranking such as BM25 so you capture both semantic relatedness and exact-token signals, and engines typically fuse scores or run a two-stage pipeline with an optional ranker, which helps especially for rare terms, identifiers, and exact phrases while keeping semantic recall high.

Modern systems accelerate search with approximate nearest-neighbor indexes and add filtering, durability, and distribution to meet production SLOs.

Vector search predates LLMs in recommendations and multimedia deduplication, and breakthroughs such as HNSW graphs and FAISS made billion-scale similarity practical.

Popular choices of vector databases include open source databases like Milvus, Weaviate, and Qdrant, the managed service Pinecone, and vector features in PostgreSQL via pgvector and in Elasticsearch via dense vectors.

These components work together to achieve RAG. Importantly, they map well to microservice boundaries, which is useful when we later deploy on Kubernetes. For instance, the vector database might be one service, the LLM another, and so on, allowing each to scale or be managed independently. Before we get into deployment, let's walk now through the two distinct phases of a RAG pipeline: *document ingestion* and *user query processing*. Understanding these two flows will make it clearer how to build and operate RAG systems.

Document Ingestion

Document ingestion is the offline process that prepares your external data so it can be used for retrieval. In this phase, we take raw documents and convert it into embeddings stored in the vector database. Think of it as building the knowledge index that your application will later query. This process can happen upfront like indexing a large corpus of company documents before the app goes live and continually as new data arrives.

A typical ingestion pipeline involves several steps shown in Figure 8-6.

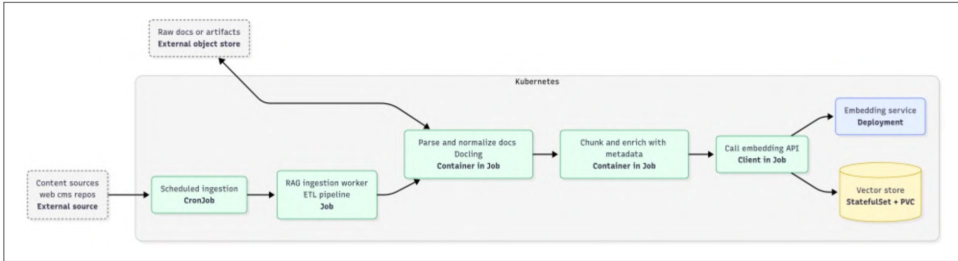


Figure 8-6. RAG Document Ingestion

Collect & parse documents

First, gather the source data you want to make available to the LLM. These could be text files, PDFs, database records, web pages, or transcripts, and you will parse each document into plain text. Often this step involves custom code or libraries to extract text from various formats. One such tool is Docling, an open-source document parsing framework designed for AI workflows that ingests heterogeneous sources such as PDFs, Word files, HTML, or scanned images and turns them into structured, machine-readable text while preserving metadata like headings or page numbers. For RAG ingestion, a consistent structured output reduces the complexity of preprocessing.

Chunk and preprocess

It is rarely ideal to embed entire documents as one piece because they may be long or cover multiple topics. Instead, documents are usually broken into chunks of a manageable size so each chunk is topically coherent and can fit within the LLM's prompt along with the question. You might split by paragraphs or headings, or use more advanced strategies such as semantic or sentence-boundary chunking to avoid breaking context mid-thought. If you use Docling, you can derive chunks directly from document structure rather than arbitrary windows, because Docling exposes sentences, paragraphs, section headers, tables, and captions as first-class elements. Docling lets you choose chunkers by sentence, by paragraph, by header-aligned sections, or by semantic grouping, and it supports overlap and maximum-size controls so you can tune recall versus prompt budget. It also emits stable identifiers and a clean metadata schema for each chunk - such as source, timestamp, version, section, and page - which preserves provenance and enables precise filtering at query time. Different chunking strategies fit different intents, with smaller, sentence-level chunks working well for FAQ-style lookup and larger, heading-aligned segments better for policies or manuals where broader context matters. Lifecycle management includes invalidating or re-embedding documents when they change, ensuring the vector store remains consistent. Metadata could include the source document title, date, author, section headings, or any tags that might be useful later for filtering or for letting the model identify the source.

Embed the chunks

Next, each chunk of text is turned into a numeric vector using the embedding model, which typically produces a high-dimensional vector. For example, you might use the Sentence Transformers model all-MiniLM-L6-v2 or call a managed API such as OpenAI's text-embedding-ada-002. This step results in one vector representation for each text chunk. Use the same model and settings for indexing and querying so the vectors remain comparable, and re-embed when you change models or tokenization. In practice, you might run this step in batches - e.g., embedding 1000 chunks at a time - to speed up the process using GPU or parallelism.

Store vectors in the database

Finally, insert each vector with an identifier and metadata into the vector database. The identifier links back to the original document or chunk so you can retrieve or display the source text. The metadata can include the chunk's raw text or a reference to fetch it from a content store. Some designs store only an ID and fetch the text on demand, while others store the text payload directly for fast retrieval; choose based on your latency and storage trade-offs. After this step, the vector database is populated with vectors representing your knowledge base and is ready to answer similarity queries.

To make this concrete, imagine a support chatbot for an e-commerce platform. Your sources might include FAQ pages, product manuals, return policies, and troubleshooting guides. In ingestion, you convert PDFs and HTML to text, chunk by section, embed each chunk, and store vectors with metadata such as source: `Product Manual X` and `section: 2.1 Installation`. After ingestion, your vector store may contain tens of thousands of vectors, each representing a piece of knowledge from your documentation.

It is worth noting that ingestion can be continuous. In a RAG system, you need a strategy to keep the vector index up to date. A Kubernetes CronJob can periodically fetch new or changed documents, generate embeddings, and upsert them, and you can also trigger event-driven re-indexing whenever a document changes so the store evolves with your data. The operational takeaway is that the vector database content is not static - it should evolve along with your data, and your platform should include the necessary jobs or processes to manage that evolution.

Now that the document chunks are in a vector database, let's look at how to use similarity queries to assemble the RAG context for answers.

User Query Processing

Once the vector database is loaded with knowledge, the RAG system can serve user queries. The query processing pipeline runs every time a new question or request comes in. Its job is to fetch the most relevant pieces of knowledge for that query,

incorporate them into the LLM's prompt, and return the model's answer. Let's break down the steps in this inference-time pipeline.

Figure 8-7 shows the user query processing pipeline, that we now breakdown in its components.

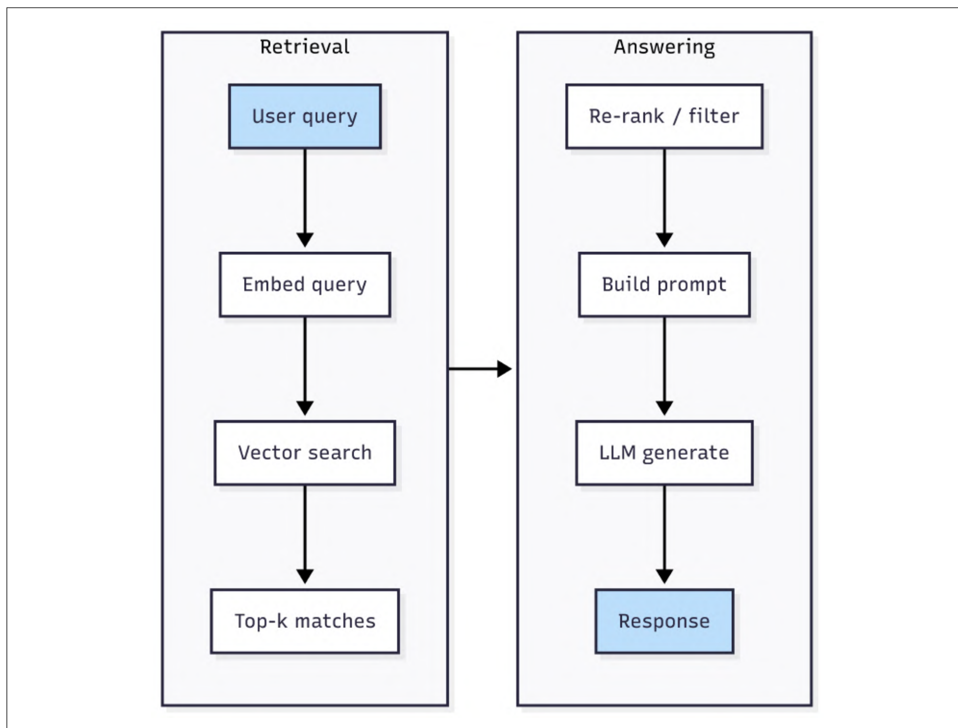


Figure 8-7. RAG user query processing pipeline

User query arrives
: A user or an upstream service asks a question or makes a request that the LLM should handle. For example, "*How do I reset my device?*". This query hits the application's API and the orchestrator component takes over.

Embed the query
: The orchestrator uses the same embedding model as in ingestion to encode the user's query into a vector. This is typically a fast operation and yields a query embedding in the same vector space as the document embeddings.

Vector search for relevant docs
: Using the query embedding, the orchestrator performs a similarity search in the vector database. The store returns the top-k nearest neighbors by cosine similarity or distance, often along with metadata and, depending on configura-

tion, the stored text payload. You can apply metadata filters or hybrid retrieval with a lexical scorer to capture rare terms, identifiers, or exact phrases while maintaining semantic recall.

Re-rank or filter results
: Optionally, a ranker scores each retrieved chunk in the context of the query so you can keep only the best few within your prompt budget. Simple heuristics like minimum similarity thresholds or recency boosts also help, and many systems do well with vector search alone when latency budgets are tight. By the end of this step you have a small set of context snippets ready to augment the prompt.

Construct the prompt with retrieved context
: The orchestrator prepares the final prompt, inserting the retrieved texts and the question into a template. Example 8-1 shows one such template. The exact wording of the prompt and how the context is presented can be tuned as needed. The key is that we ground the model by giving it facts to work from, for example a paragraph from the manual that contains the reset steps. In our example, the context might include a paragraph from the manual about resetting, which contains the specific steps.

LLM generates an answer
: The orchestrator sends the composed prompt to the LLM service via an API call to the model inference server. The LLM processes the prompt and produces a completion - in this case, hopefully a step-by-step answer explaining how to reset the device, drawn from the provided context. Because we included the relevant snippet, the model doesn't have to invent facts; it just has to articulate the answer in natural language. The output for our example might be something like: "*To reset your device, first hold down the power button for 10 seconds until the LED blinks*“, which mirrors the documentation and is formulated by the LLM.

Post-process and return the response
: The orchestrator post-processes the model output before returning it to the caller. Typical steps include formatting, attaching source citations from metadata, enforcing guardrails, and truncating to size limits. The final answer is delivered back through the API or UI.

Example 8-1. Example template to build up the prompt from RAG documents

```
Use the following context to answer the question.
If the context doesn't have the answer,
say you don't know.

Context:
{retrieved_text}            ❶
```

```
Question: {user_question}  ❷
Answer:
```

❶ Placeholder replaced by the documents retrieved from the vector store

❷ Parameter replaced with the actual user query

From the user's perspective, this pipeline is invisible, and they simply get a helpful answer that references the right information. Vector search is usually fast enough that LLM generation dominates latency, so a well-implemented RAG pipeline still feels real time. If no strong context is found, the orchestrator should abstain gracefully rather than risk a hallucinated answer. With these steps in place, the LLM's response is grounded in your knowledge base and remains aligned with up-to-date facts.

Now that we have all the ingredients of a RAG system, let's see how we can map the individual RAG components to Kubernetes primitives.

RAG on Kubernetes

Let's map the components from "RAG Components" on page 219 onto Kubernetes and show how to operate them as one production-grade system. A production RAG stack is a set of cooperating services with distinct lifecycles and SLOs that fit cleanly into Deployments, StatefulSets, Services, and Jobs. Kubernetes lets you scale each piece independently, roll out safely, and standardize configuration and security across environments.

Table 8-1 give a quick overview of the various RAG specific components, their associated K8s workload type and their anticipated resource requirements.

Table 8-1. Overview of RAG components deployed in Kubernetes

Component	K8s Primitive	Type	Resources	Storage
Vector database	StatefulSet + PVC	Stateful	High RAM/CPU, fast volumes	Persistent volumes
Embedding	Deployment / Sidecar / In-process	Stateless	CPU for light models; GPU optional	None
Orchestrator/API	Deployment + Service (+ Ingress)	Stateless	CPU and moderate RAM	None
Ingestion	CronJob / Job	Batch	CPU; GPU optional	Reads/writes vector store

Now let's examine Kubernetes support for each component in more depth and, where it adds clarity, point to the associated patterns in Kubernetes Patterns.

Vector database

The vector store is the backbone of retrieval, and on Kubernetes it belongs in a StatefulSet with a PersistentVolumeClaim per replica so shards have stable identities and data persists across restarts. If the vendor offers an operator, adopt it to encapsulate cluster setup and upgrades, and size memory so hot indexes stay resident while enabling approximate nearest neightbor (ANN) indexes for scale. Expose it on a cluster-internal Service and use NetworkPolicies to restrict which Pods can connect, then treat it like any critical database with snapshots and tested restores. This maps directly to the *Stateful Service* and *Service Discovery* patterns.

Embedding service

Embedding models can be deployed in different ways depending on your performance and operational needs. The most common production setup is to serve embeddings through a lightweight model server packaged as a Deployment, which allows you to scale it independently and allocate either CPU or GPU resources as needed. For small, efficient models it may be simpler to embed directly in-process within the orchestrator, avoiding a network hop and keeping latency low. A middle ground is a Sidecar in the orchestrator Pod to share fate while versioning the model independently, as shown in Figure 8-8. The key is consistency between ingestion-time and query-time encoding, and the Sidecar pattern fits well when you need a local helper without coupling builds. Some databases can generate embeddings in-database at write or query time - for example, Weaviate's vectorizer modules can embed on write operations and at query, and Postgres with pgvector can drive automatic embedding via SQL triggers and extensions. This keeps ingestion and retrieval encoders aligned but increases coupling between the database and model choice. Whichever pattern you use, enforcing a single embedding model and configuration for both ingestion and query-time encoding is crucial.

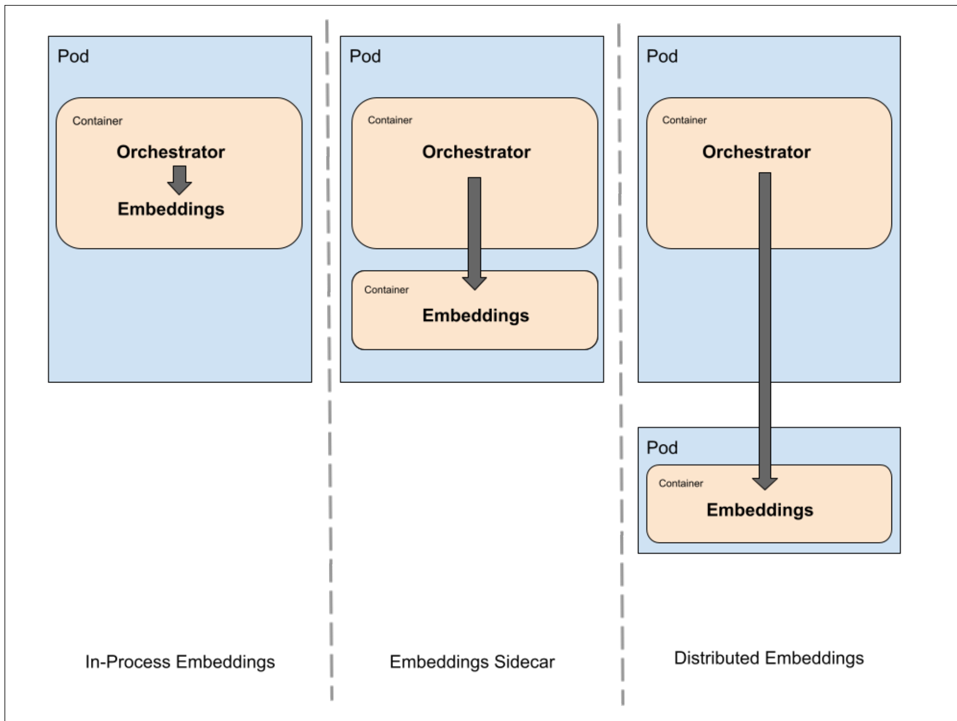


Figure 8-8. Multiple ways to deploy an embeddings service

Orchestrator
: The orchestrator is the "brain" that encodes the query, retrieves, optionally reranks, constructs the prompt, and calls the LLM, so run it as a stateless Deployment exposed through a ClusterIP Service or, if it faces end users, through an Ingress or API Gateway. Because it coordinates multiple dependencies, the orchestrator should be carefully instrumented: propagate trace context across calls so you can measure end-to-end latency and identify bottlenecks. Configuration such as prompt templates, thresholds, or retrieval parameters should live in ConfigMaps so you can tune them without code changes, and credentials should be mounted from Secrets to keep them safe. The *Secure Configuration* pattern has more information how you can harden secret configuration. Scale with the horizontal pod autoscaler for steady load and use Knative or KEDA when you need event-driven bursts or scale-to-zero for idle paths. Checkout the *Elastic Scale* pattern to learn more about Knative and KEDA, and how it supports up and down scaling, including scale to zero.

Re-ranker (optional)
: If precision matters, run a cross-encoder or heavier reranker as its own Deployment and call it selectively for high-stakes queries to balance cost and latency.

Simple heuristics can stay in the orchestrator, but a separate service lets you tune resources and release cadence independently, following the Stateless Service pattern. Keeping the re-ranker optional allows you to balance cost and quality, enabling you to switch it on for high-stakes queries while letting most traffic flow through the faster path.

Batch ingestion jobs
Document ingestion is not a one-time event but an ongoing process, and Kubernetes is well-suited to running this work in the background. You can schedule ingestion with CronJobs that periodically fetch new or updated sources, parse, chunk and embed them, and upsert results into the vector database. For near-real-time pipelines, event-driven Jobs can be triggered by file uploads or database updates. Document ingestion can be also modelled nicely as an endpoint of an Event Mesh as offered by Knative Eventing. Use resource requests and limits so that ingestion workloads do not starve user-facing services, and separate the namespaces or node pools if you need stricter isolation. By treating ingestion as a first-class workload, with monitoring and retries, you ensure the vector database fresh and your RAG system reflects the latest state of your domain. Technical details can be found in the *Batch Job* and *Periodic Job* patterns.

With RAG we saw how to ground LLMs in trusted knowledge and operate the supporting components on Kubernetes We now turn to agentic workflows, where the model not only consumes context but also plans actions, chooses tools, and iterates toward goals in short think-act-observe loops.

Agentic Workflows

Agentic apps wrap the inference calls to a model in a small control loop that can plan, call tools, observe results, and iterate until a goal is met.

In general the control loops looks like Figure 8-9 and contains the following steps.

Perceive
Read new signals: user input, tool output, and conversation state.

Think
Plan the next step, decide whether a tool is needed, and shape the next prompt turn.

Act
Execute an action: call a tool, run code, fetch data, or draft a candidate answer.

Observe
Capture the tool result or user follow-up and normalize it to the working context.

Reflect
: Check progress against the goal, revise plan, and decide to stop or continue.

Remember
: Store short-term scratchpad items and long-term facts in external memory.

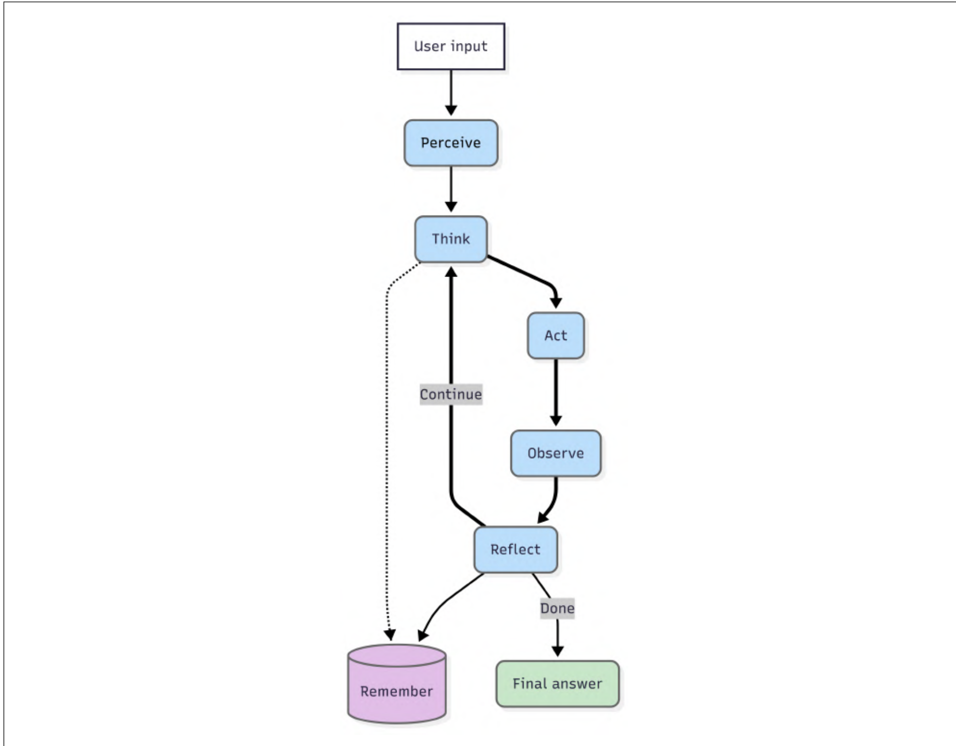


Figure 8-9. Agentic control loop

This flow is a refinement of the well-known *ReAct loop* that we describe in "The ReAct loop" on page 230. This loop applies for simple agents and build the foundation for more complex scenarios involving *multi-agents* described in "Multi-Agent Systems" on page 238 and *ambient agents* that we explore in "Ambient Agents" on page 241.

The ReAct loop

The ReAct pattern interleaves *chain-of-thought* (CoT) reasoning with tool actions so the model can think, act, observe, and repeat in a compact loop. Originally introduced by Yao et al. in the paper ReAct: Synergizing Reasoning and Acting in Language Models, ReAct showed that models reduce hallucinations and improve success rates when they can alternate between reasoning and calls to external sources

such as a search API. In a ReAct agent, the LLM emits intermediate thoughts and, when needed, an action with JSON arguments, the tool executes, the observation is appended to context, and the model continues until a stop condition. In production you do not expose CoT to users; you keep traces internal or replace them with summaries, but the control flow stays the same. Log each thought, action, and observation as structured records so you can debug behavior later and correlate cost with quality.

Let's focus for the moment on the *Act* part of the agentic control loop, as it's the crucial step that allows to include information that is not part of the model's training data either because it is too new or domain specific information not accessible for the model training. These actions are commonly refered to as *tools* and can be anything from simple web searches to API calls to enterprise internal backend services. Tool use comes in two execution paths that you can mix in one workflow.

Client-executed function tools
: The model emits a function call with JSON arguments; your client code performs the action and posts the result back, keyed by a call_id. This is the portable baseline for fine-grained control and audit in your control plane. Example 8-2 shows the request-response flow in which the server asks in its resposne the client to call a tool from the client-side and send the result back to the server. This kind of multi-step interactions requires the agentic flow to be *stateful* to keep the conversation history. Client-side tool calling turned out to be quite fragile, as every agentic framework expects the tool call request to be in a different format.

Server-executed tools
: The runtime executes tools on your behalf, including remote Model Context Protocol (MCP) servers. We will dive into the Model Context Protocol standard in ??? as it is the de-facto standard these days for tool interaction. MCP allows for a much better integration of domain knowledge into the agent flow than client-side tool calling. For now it is good enough to know that MCP is a protocol that facilitates tool calling and discovery substantially on the server side.

Example 8-2. Client-side function calling with OpenAI's Responses API

```
# Initial request including description of available tools
curl https://api.openai.com/v1/responses \                ❶
  -d '{
    "input": [
      {"role": "user", "content": "Do I need an umbrella in Berlin today?"}
    ],
    "tools": [
      {
        "type": "function",                               ❷
        "name": "get_weather",
        "description": "Get the weather information for a city and ISO date.",
```

```
        "parameters": {
          "type": "object",
          "properties": {
            "city": { "type": "string" },
            "date": { "type": "string", "format": "date" }
          },
          "required": ["city", "date"],
          "additionalProperties": false
        },
        "strict": true
      }
    ],
    "tool_choice": "auto"
  }'
# Part of the returned response, asking the client for a a tool call
{
  ...
  "output": [
    ... ,
    {
      "type": "function_call",   ❸
      "call_id": "call_wx_1",    ❹
      "name": "get_weather",
      "arguments": "{\"city\":\"Berlin\",\"date\":\"2025-09-20\"}"
    }
  ],
  "status": "incomplete"
}

# Part of the second client request,
# holding the result of the tool call
curl https://api.openai.com/v1/responses \
  ...
  -d '{
    "input": [                   ❺
      ...,
      {
        "type": "function_call_output",
        "call_id": "call_wx_1",
        "output":
          "{\"precipitation_chance\":0.80,
            \"summary\":\"Heavy rain expected in the afternoon.\"}"
      }
    ]
  }'
```

❶ Initial user query.

❷ Definition of a client-side tool.

❸ Response by the server asking the client to call a tool.

❹ Correlation id to connect the tool call response with its request.

❺ Second request returning the result of the tool call.

Originally, client-side tool calling has been performed by the caller of the agentic loop so that he takes over the responsibility for the actual call. In this case, when the LLM decides in its reasoning phase that a tool need to be called, based on the description and meta data that a tool exposes, it returns control back to the caller, asking him to call the tool and to return the results in the next step of this multi-turn conversation.

The rest of this chapter builds from here. In "OpenAI's Responses API" on page 234 we dig into the Responses features for state, events, and approvals. In "Agentic Frameworks and Runtimes" on page 233 we compare client-side libraries with server-side runtimes. In "Multi-Agent Systems" on page 238 we scale the loop across teams of agents, and in "Ambient Agents" on page 241 we make the loop event-driven on Kubernetes.

Agentic Frameworks and Runtimes

Building an agentic workflow from scratch is challenging, so frameworks and runtimes simplify the job. Broadly, you will see client-side agentic libraries embedded in your code and server-side agentic runtimes exposed as services. We won't go into to many details here as we focus on operating agentic systems on Kubernetes. However, for this it is important to understand how to classify the multitude of frameworks.

Client-side agentic frameworks
: These libraries help you to run the loop inside your own applications, giving you full control and easy debugging at the cost of managing orchestration. LangChain provides abstractions for prompts, memories, and tool use across Python and JavaScript, with a broad ecosystem for web search, databases, and REPLs. LangGraph models agent steps as a graph, making branching and concurrent sub-tasks explicit and observable. crewai focuses on multi-agent collaboration via role-based agents that message and delegate, which helps when specialization and parallelism pay off. Because these libraries live in your runtime, you own the loop that calls the LLM, executes tools, feeds results back, and decides when to stop, which maximizes control while increasing complexity. In production, even these "client-side" frameworks typically are leveraged by the orchestrator in containerized microservices on Kubernetes, subject to the same packaging, scaling, and observability practices you use elsewhere.

Server-side agentic runtimes
: Bespoke backend services encapsulate the loop behind an API so a client sends one request and the backend performs multi-turn reasoning and tool use. For example, OpenAI's Responses API provides stateful multi-turn interactions, inte-

grated tool usage, structured outputs, event streaming, and pause-and-resume for human-in-the-loop, so you do not have to write the orchestration loop yourself. You learn more about the Responses API below in "OpenAI's Responses API" on page 234. The Responses API supports server-executed tools, including remote MCP tools, as well as client-executed function tools when you need to keep actions in your control plane. Llama Stack offers an open, self-hostable runtime with both an Agents API and an OpenAI-compatible endpoint, including a Responses-style flow, so you can run agentic backends on Kubernetes with your model choices. By contrast, the vLLM project works on an OpenAI-compatible server with tool calling and structured output support, and you should check the current documentation for feature parity with the Responses API over time.

The practical distinction is where the agent's orchestrator runs. Server-side runtimes hide the loop behind a network API, which simplifies client code and centralizes scaling and governance, while client-side frameworks keep logic local for maximum customization and composability. In practice you can mix both: use LangChain in your app while targeting a Llama Stack backend for inference and server-side tools, or keep tools local as client-executed functions even when planning happens server-side.

OpenAI's Responses API

OpenAI's Responses API is designed for agentic workflows in a single, stateful API call. The Responses API introduced features that simplify agent development: automatic conversation state across turns, structured outputs, integrated tool usage, streaming of intermediate tool events, and better error handling built in.

You send the user input and a catalog of tools with JSON Schemas, and the service can autonomously sequence tool calls, feed observations back into the model, and return a final answer. Two execution paths can coexist in one flow. *Server-side tools* run within OpenAI's runtime, including tools accessed via the *Model Context Protocol* (MCP), and you receive streamed events and the final answer without implementing the loop client-side. We have to say more about MCP in ???. *Client-executed function tools* let the model emit a tool call with name and JSON arguments, your service performs the action, and you resume by posting the result so the model can continue and finish.

Importantly, the Responses interface is rapidly becoming a de facto standard because several backends implement or target compatible endpoints. Meta's Llama Stack ships an OpenAI-compatible interface and a Responses implementation that is usable but still in active development, which enables self-hosted agent runtimes on Kubernetes without changing client code. vLLM has introduced a Responses entry point in its OpenAI-compatible server and is progressing rapidly as of mid-2025 so it's definitely worth it to checkout this work. OpenAI's own cookbook and community guides also

reference vLLM offering a Responses-compatible API, underscoring the ecosystem's convergence on this contract. In parallel, LiteLLM provides a proxy that exposes a `/responses` endpoint and routes to multiple providers, giving teams a compatibility layer while the various servers continue to mature.

The takeaway is portability: you can standardize client code on the Responses API while choosing where to run the agentic loop - OpenAI's cloud, a self-hosted Llama Stack, or a vLLM-based service on your cluster - and swap as your operational needs evolve.

Human-in-the-loop fits naturally into this flow. You can pause on model-requested actions to ask a user for approval, collect additional inputs, or escalate to a reviewer before resuming, and you can enforce approval gates for sensitive tools so the model cannot proceed until you confirm. When using remote providers via MCP, the API can surface explicit approval requests for those calls, which gives you an auditable checkpoint before any side effect happens.

In short, Responses provides *agentic reasoning* as a service while letting you control which tools exist, which calls execute on your side, and when to require approval, and the growing set of compatible backends makes it a pragmatic choice for portable agent architectures.

Agents on Kubernetes

In "Agentic Frameworks and Runtimes" on page 233 we categorized popular libraries and API services you can use to implement an agentic workflow. In this section we focus on deployment models for agent-enabled applications on Kubernetes and show what Kubernetes-native integrations look like in practice. We keep this high level and refer to ??? for deeper operational details.

Kubernetes is a natural home for agentic systems because it gives you composable building blocks for the orchestrator, the tools, and the memories.

Now let's turn to Kubernetes-native integrations that bring agents into the control plane via Custom Resource Definitions and controllers. As of mid-2025 this space is evolving quickly, but one of the more mature projects is Kagent, originally started by Solo.io and growing in the Cloud Native Computing Foundation (CNCF) community. Kagent is a Kubernetes-native operator that lets you declare agents, tools, and exposure modes as custom resources and then reconciles them into runnable Pods. It leans into protocol compatibility for tools and agent-to-agent exchange, so you can register MCP-compatible tool servers and expose A2A skills without leaving the control plane. You manage agents with the same GitOps and security practices you already use for Deployments and Jobs.

The trimmed example in Example 8-3 shows the intent: define the reasoning loop, attach tools via MCP, and publish an A2A skill, while the operator handles Pods, configuration, and status.

Example 8-3. Example of an Kagent agent definition

```
apiVersion: kagent.dev/v1alpha2
kind: Agent
metadata:
  name: k8s-a2a-agent
  namespace: kagent
spec:
  description: An example agent
  declarative:
    modelConfig: default-model-config          ❶
    systemMessage: |
      You are a helpful Kubernetes agent.      ❷
    tools:                                     ❸
      - type: McpServer                        ❹
        mcpServer:
          name: kagent-tool-server
          kind: RemoteMCPServer
          toolNames:
            - k8s_get_resources
  a2aConfig:                                    ❺
    skills:
      - id: get-resources
        name: Get Resources
        inputModes: ....
        outputModes: ....
```

❶ Reference to agent configuration.

❷ System prompt.

❸ List of tools to use.

❹ Reference to an MCP server declared in a different resource RemoteMCPServer.

❺ Configuration specific for connecting via Google A2A protocol.

Emerging Kubernetes agent experiments

Two efforts in mid-2025 are worth tracking as they move toward production readiness. First, the Kubernetes community's agent sandbox explores a controller and custom resource for isolated, stateful, singleton-style runtimes with stronger boundaries, persistent identity, and hibernation and resume. The goal is to support interactive or

untrusted agent workloads that benefit from VM-like isolation yet stay manageable as Pod-shaped resources. Second, Kagenti positions itself as framework-neutral middleware with an operator and a uniform surface for agents, aiming to standardize identity, configuration, and exposure while integrating protocol bridges such as MCP and A2A. Treat these as experiments (as of 2025) and evaluate their APIs and operational fit as they mature.

Not every Kubernetes agent integration leverages custom resources to define the agentic workflow. One notable example here is Llama Stack which is a general purpose API layer for agentic applications that has support for agentic flows including tool calling and multi turn reasoning. Llama Stack can still leverage an operator for managing the installation, but otherwise it relies on custom configuration files for configuring the backend systems that it's using to implement the agentic functionality.

Most agentic platforms converge on similar Kubernetes concepts despite their different approaches.[2]

- In all most agentic platforms, **custom resources** and **controllers** extend Kubernetes to manage agents declaratively. Platforms like Kagent or Kagenti introduce CRDs that model agents as resources, letting you manage them with the same GitOps workflows you use for Deployments. Controllers reconcile these resources into Pods and Services, bringing infrastructure-as-code benefits to AI systems. (*Controller*, *Operator*, *Declarative Deployment*)
- Long-lived **stateful pods** maintain conversational context across interactions. Unlike stateless services, agents often run as singleton Deployments or single-replica StatefulSets to preserve session state. When scaling is needed, platforms either shard sessions across pods or externalize state to enable round-robin load balancing, borrowing patterns from stateful microservices. (*Singleton Service*, *Stateful Service*)
- **Batch jobs** offload discrete tasks from the main agent loop for heavy lifting. When an agent needs to generate a report or process large datasets, it can submit a Kubernetes Job and await the result. This separation brings automatic retries and resource isolation, similar to how ML pipelines decompose work. (*Job*, *Periodic Job*)
- **Event-Driven Architecture (EDA)** enables ambient agents that respond to system changes. In EDA setups agents are deployed as listeners that react to Kafka topics, Kubernetes events, or webhooks. Combined with a scaling platform like Knative Eventing or KEDA, these agents can scale from zero when idle and

2 We've added the corresponding pattern from *Kubernetes Patterns* in parentheses where applicable.

burst when events arrive, treating agents as reactive microservices rather than request-response endpoints. We talk more about ambient agents in "Ambient Agents" on page 241. (*Elastic Scale*)

- **Tool integration** is essential for agentic apps as it exposes capabilities through standard APIs via Services. Tools run as separate Deployments with ClusterIP Services, and agents call them by DNS name. Many platforms adopt MCP to standardize these interfaces, allowing any MCP-compliant tool to work with any compatible agent framework. In ??? we go into much more details of how to operator MCP servers on Kubernetes. (*Service Discovery*, *Declarative Deployment*)
- **Persistent Storage** for memory ensures agents retain knowledge across restarts. Vector databases for long-term memory run as StatefulSets, while conversation history lives in databases with PersistentVolumes. This externalization makes ephemeral agent pods viable for stateful AI processes, following the same patterns as any data-dependent microservice. (*Stateful Service*)
- Native **Kubernetes security** controls what agents can access and do. ServiceAccounts with restricted agent permissions, NetworkPolicies sandbox network access, and Secrets mount credentials with least privilege. Multi-tenant deployments isolate agents in separate Namespaces, while admission controllers enforce resource quotas and policy compliance. (*Process Containment*, *Secure Configuration*, *Access Control*, *Network Segmentation*)
- **Observability** stacks treat agents as measurable services. Agentic apps expose metrics for token counts and tool calls, stream verbose reasoning logs for debugging, and can generate Kubernetes Events for significant actions. When properly configured, this allows SREs to monitor agents with the same dashboards and alerts used for other services.

In practice, successful agent deployment on Kubernetes combines robust containerization, careful state management, appropriate resource allocation, and comprehensive observability.

The platform becomes your control plane for agentic AI, managing lifecycle and resources while agents focus on reasoning and tool orchestration. Whether you deploy a simple ReAct loop in a single container or coordinate multi-agent cohorts like we describe next in "Multi-Agent Systems" on page 238 across namespaces, Kubernetes provides the scheduling, networking, and storage primitives to run agents reliably at scale.

Multi-Agent Systems

Multi-agent systems assemble several specialized agents that collaborate toward a goal larger than any one agent could deliver. Each agent is an autonomous service with its

own prompt, tools, and guardrails, and it accesses one or more LLMs through remote APIs. This collaboration creates useful side effects: agents pass intermediate results, cross-check each other's work, and parallelize subtasks to improve both quality and throughput. Agents are scoped to independent tasks so responsibilities stay clear and coupling remains low. For example, think of a software team: a *Planner* breaks work into steps, a *Coder* drafts changes, and a *Tester* verifies behavior before a final *Reviewer* signs of[3]. Specialization lets each agent focus on a narrow competency while the system as a whole moves faster and with more confidence. A major benefit of this architecture is that each specialist operates with a much smaller working context than a single monolithic agent would need for the whole problem, which makes prompts more focused, reduces token usage, and improves accuracy. These agents coordinate through an explicit control flow so their partial results compose into a coherent outcome.

The heart of a multi-agent system is its coordination logic. One common pattern is a central orchestrator that assigns work to role agents and aggregates outcomes, which is the shape you see in crew-style frameworks where a facilitator routes coding questions to a coding agent and compliance questions to a policy agent. Figure 8-10 show this setup with a central planning agent conducting multiple worker agents.

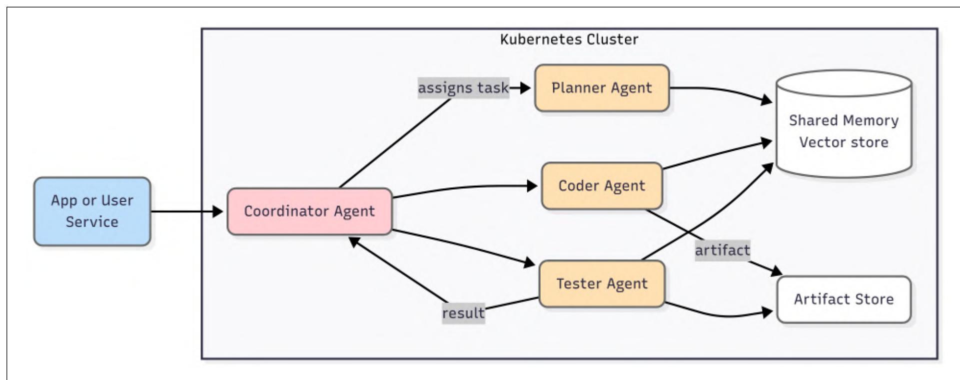


Figure 8-10. Agents orchestrated by an coordinatork.

An alternative is peer-to-peer coordination in which agents message one another directly, discover capabilities dynamically, and escalate or delegate without a single hub, as shown in Figure 8-11. Google's A2A protocol discussed in ??? formalizes this style by standardizing discovery, capability exchange via an agent card, task lifecycles, and artifact streaming across agent boundaries, which enables interop across teams and vendors.

3 There are several projects that directly support this multi-agent coding flow. One popular option as of mid 2025 is Claude-Flow, a sophisticated multi agent setup using Claude as backend model

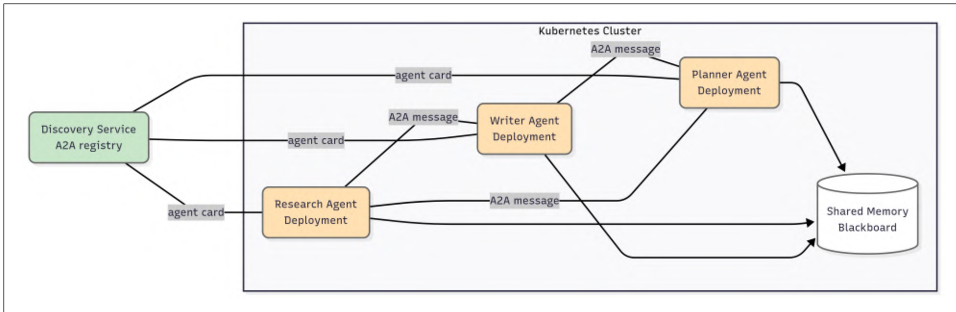


Figure 8-11. Agents triggering each other on demand.

In both models, the system succeeds or fails on the discipline of its messages - what gets shared, when, and with which guarantees - rather than on any individual prompt.

On Kubernetes, you typically model each agent as a Service-backed Deployment and connect them synchronously over HTTP or gRPC, or asynchronously via a pub/sub messaging fabric. Alternatively, multiple agents can run in a single Pod when a framework coordinates agent dialogue in process. This arrangement simplifies cross-agent state sharing and reduces latency, but it couples lifecycles and scaling so every agent scales together, which limits elasticity and is rarely a fit beyond small or tightly bound teams. For the rest of this section, we focus on a distributed design where agents collaborate over the network. Shared memory backends provide the glue for collaboration, using a vector store, a document store, or a blackboard where agents post findings, pending tasks, and artifacts for others to consume. This shared state lets the system remember what happened across agent boundaries while still isolating each agent's runtime and quota. The usual platform concerns still apply - service discovery, retries, backoffs, and circuit breakers - because agents are distributed systems in miniature. We will tie these pieces back to the protocols in ??? so you can choose message shapes that survive versioning and team boundaries.

A concrete example of a multi-agent system is one for customer support automation: one agent monitors incoming support tickets (as an ambient trigger agent), it then delegates each ticket to an appropriate specialized agent - say, a NetworkTroubleshooter agent or a BillingInquiry agent - and finally a Summary agent compiles a report of what was done. The coordination logic here decides which specialist agent gets involved and when the process is done.

Multi-agent systems shine in such scenarios, but they also introduce complexity in ensuring all agents work in harmony and don't step on each other's toes. Careful design of roles, communication channels, and fail-safes (like what if two agents disagree?) is required.

In summary, multi-agent is collaborative intelligence. You compose small, sharp agents and add a coordination layer - centralized or peer-to-peer - and you back them with shared memory that preserves context and evidence. Done well, this is *agent orchestration* in the literal sense: many instruments, one score, and clear cues.

With that foundation in place, we now turn to ambient agents and the background services that watch event streams, detect conditions, and trigger workflows, to see how they complement multi-agent designs on Kubernetes.

Ambient Agents

Ambient agents run continuously in the background and react to signals from their environment rather than waiting for an interactive prompt. They live alongside your systems and take action when triggers fire - a new file appears, a row changes, a sensor crosses a threshold, or a timer goes off. Think of them as passive until needed: they do not start conversations, but they do not need a human to ask before they act.

A practical example is a Kubernetes caretaker that monitors cluster health signals for crash loops or CPU pressure and immediately investigates by querying logs and comparing recent metrics. If the findings match a known pattern, the agent attempts a targeted remedy like restarting a Deployment, rolling back a config, or scaling out a Service, and only escalates to a human when automated actions fail or when policy marks the situation as high risk. The involved components of such an ambient agent setup is shown in Figure 8-12.

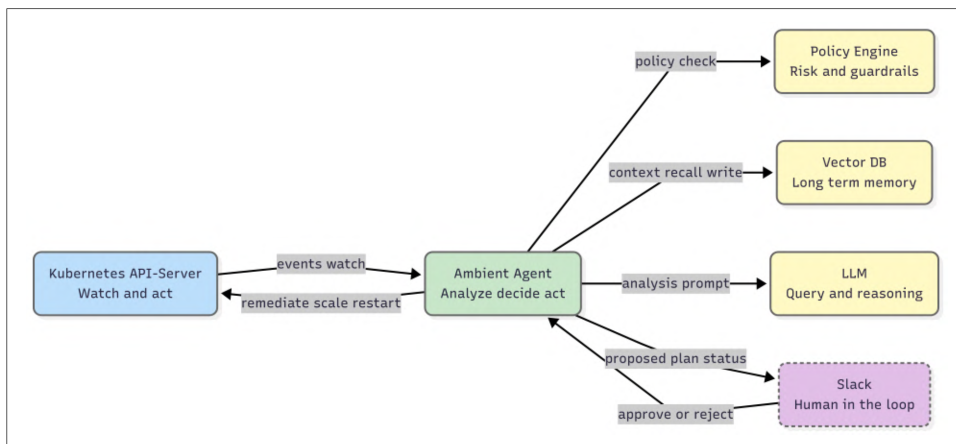


Figure 8-12. Ambient agent example watching on Kubernetes events

Ambient agents are built on *event-driven architecture* (EDA). They subscribe to queues, webhooks, or file watchers, update their working context, decide whether to act, and then call tools. For sensitive operations they insert human-in-the-loop (HIL) checkpoints: the agent drafts a plan, routes it to an approver, and executes only

after an explicit “go”. You can tune autonomy by policy - recommend only, approve to act, or auto for low risk - and you can bound variation with a determinism budget so replays and retries behave predictably. On the output side, every action should leave an evidence trail with inputs, decisions, and artifacts so operations remain auditable.

Human in the Loop

Human in the loop is a deliberate checkpoint where a person reviews an agent’s plan or outcome and explicitly authorizes the next step before the agent proceeds. You use it for high-risk or irreversible actions, when policy demands human oversight, or when signals are ambiguous and confidence is low. Typical examples include pushing a production hotfix, rolling back a config that could cause downtime, approving a large financial transaction, or sending a high-volume customer notification. Feedback can be gathered through chat and messaging systems like Slack or Teams where the agent posts the proposed plan and waits for an *approve* or *reject* reply. A more decoupled pattern emits an approval request on a message bus with a correlation ID, then listens for the corresponding decision event, possibly emitted by an dedicated UI. For auditability, the agent should attach its rationale and diffs to the request, record the approver and decision, and post the final result after execution. This keeps autonomy where it is safe and moves judgment to humans where it matters most.

For ambient agents, the platform concerns are the same as for any distributed system: service discovery, retries and backoffs, circuit breakers, idempotent handlers, and clear ownership of configuration and secrets.

In practice, you will get the best results when ambient agents blend three disciplines: reliable event handling with idempotent actions, explicit human checkpoints for irreversible changes, and clear Kubernetes ownership boundaries for scaling and security. This keeps ambient agents predictable like any other microservice while giving you the superpower of proactive operations at scale.

Lessons Learned

AI-driven applications on Kubernetes work best when we draw clear lifecycle boundaries between orchestration, inference, and state. Treat the LLM server as a replaceable dependency, keep application logic in its own Deployment, and let data systems own their StatefulSets and backups. This separation lets you upgrade, scale, and troubleshoot each part independently while keeping SLOs predictable.

Two dominant shapes recur in the wild and should guide design choices.

- **Interactive chat-style** apps run a synchronous request path where latency is king, so pre-warm the LLM, keep the orchestrator lean, and minimize round trips.

- **Backend event-driven** services run asynchronously inside your microservice mesh, where idempotency, buffering, and eventual consistency matter more than raw response time.

Batch jobs, continuous control loops, and tool-driven automations sit alongside these cores and give you a cost lever by shifting non-urgent work off the hot path.

RAG succeeds when ingestion and query-time pipelines stay consistent and observable. Use one embedding model for both phases, choose chunking that matches your content, and store provenance so you can cite and filter confidently. Vector databases belong in StatefulSets with snapshot and restore plans, while ingestion should run as Jobs or CronJobs that do not starve user traffic. A reranker can boost precision for high-stakes queries, but keep it optional so you can trade cost for quality per route.

Agentic workflows add a small, explicit control loop around the model and make tools first-class citizens. Decide early which tools execute client-side for maximum control and which execute server-side for simplicity, and standardize interface contracts so you can swap runtimes without rewriting clients. Human in the loop is not an afterthought; use approval gates for risky actions, capture rationale and artifacts, and make every step auditable. Portability improves when you target a stable request contract and keep tools discoverable through a protocol or registry rather than ad hoc glue code.

For architects, the most important concern for AI apps is composability. Design around protocols and SLOs, not vendors, put retrieval and tools behind stable services, and pick sync or async flows based on user experience and failure modes.

For Kubernetes administrators, the mandate is safe, observable, and right-sized runtime. Map stateless services to Deployments and scale them elastically, run vector stores and durable memories in StatefulSets with tested backups, and schedule ingestion as Jobs. Guard the blast radius with RBAC, Secrets, and NetworkPolicies, isolate GPU pools for inference, and instrument every hop with tracing so you can see where tokens and milliseconds go. Adopt operators where available, apply resource requests and limits consistently, and use event-driven autoscaling to match bursty workloads without paying for idle capacity.

With these patterns and boundaries in hand, we are ready to go deeper in the how. In the next chapter we turn the high-level designs into production guidance and show how to stand up agentic applications on Kubernetes and dive into some more tricky challenges like securing MCP and A2A communications.

About the Authors

Dr. Roland Huss is a seasoned software engineer with over 25 years of experience in the field. Currently working at Red Hat, he is the architect of OpenShift Serverless and a former member of the Knative TOC. Roland is a passionate Java and Golang coder and a sought-after speaker at tech conferences. An advocate of open source, he is an active contributor and enjoys growing chili peppers in his free time.

Daniele Zonca is a Senior Principal Software Engineer and Architect for model serving of Red Hat OpenShift AI, Red Hat's flagship AI product combining multiple stacks.