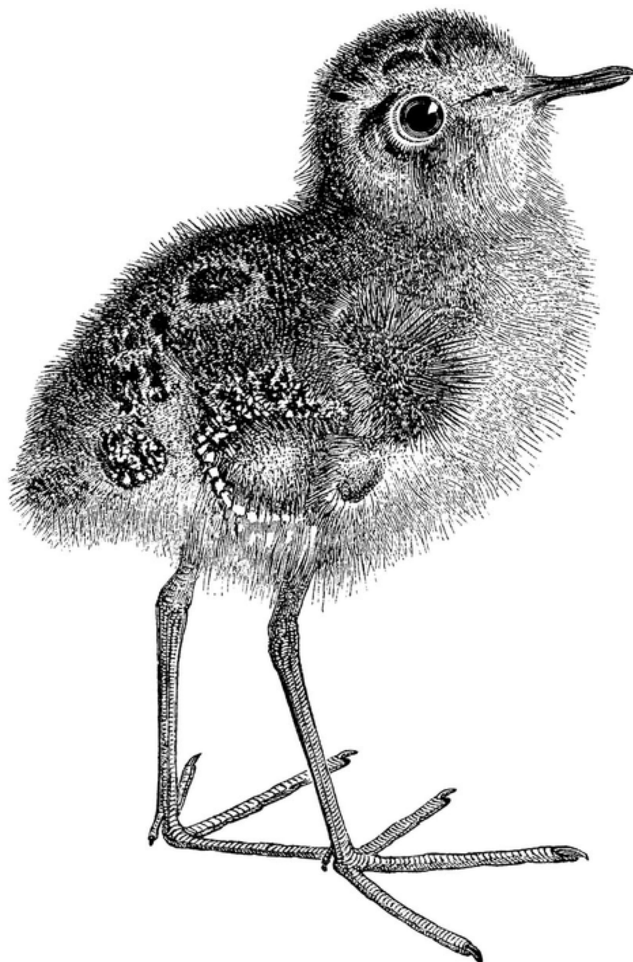
O'REILLY®

Generative AI on Kubernetes

Operationalizing Large Language Models



Early Release

RAW & UNEDITED

Roland Huss
& Daniele Zonca

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

- May 2025: First Edition

Revision History for the Early Release

- 2025-03-12: First 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]

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: Running 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.

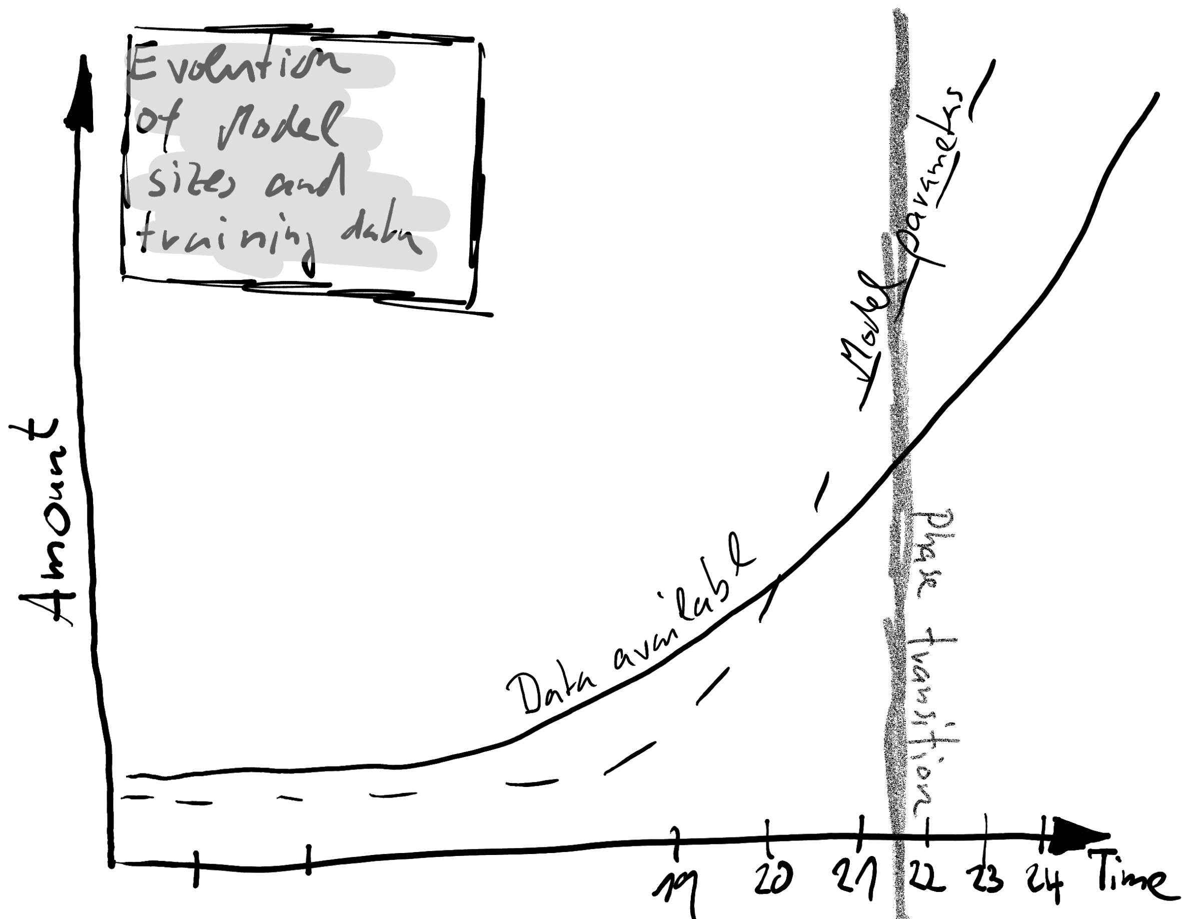


Figure 1-1. Explosion of parameters and training data lead to a phase transition.

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 CrashLoopBack 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 configuration 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.

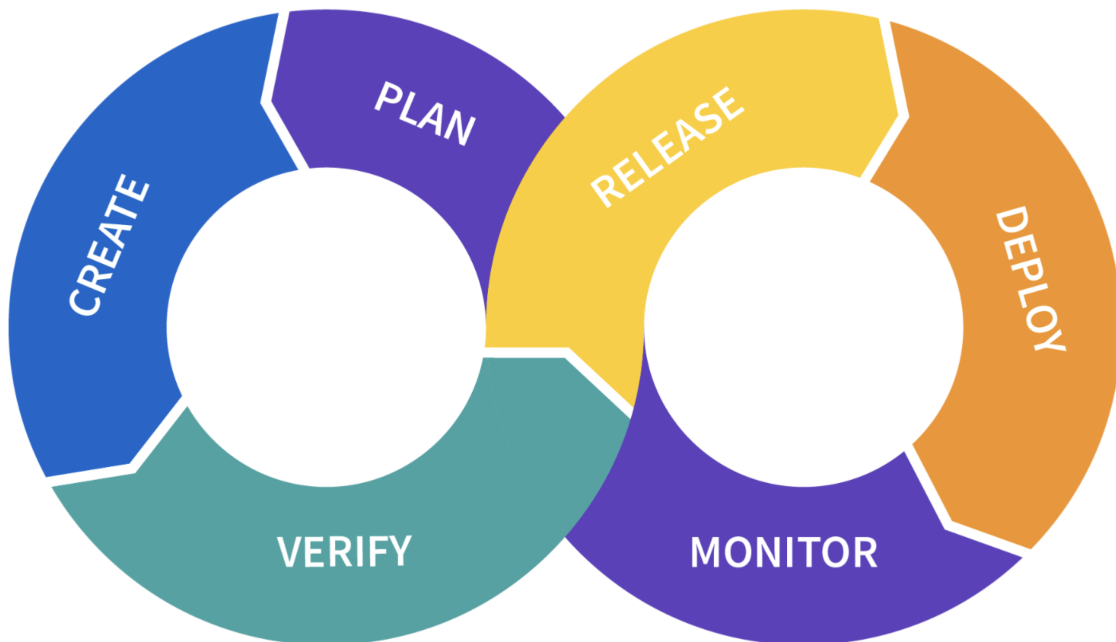


Figure 1-2. The inifinite DevOps loop

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.

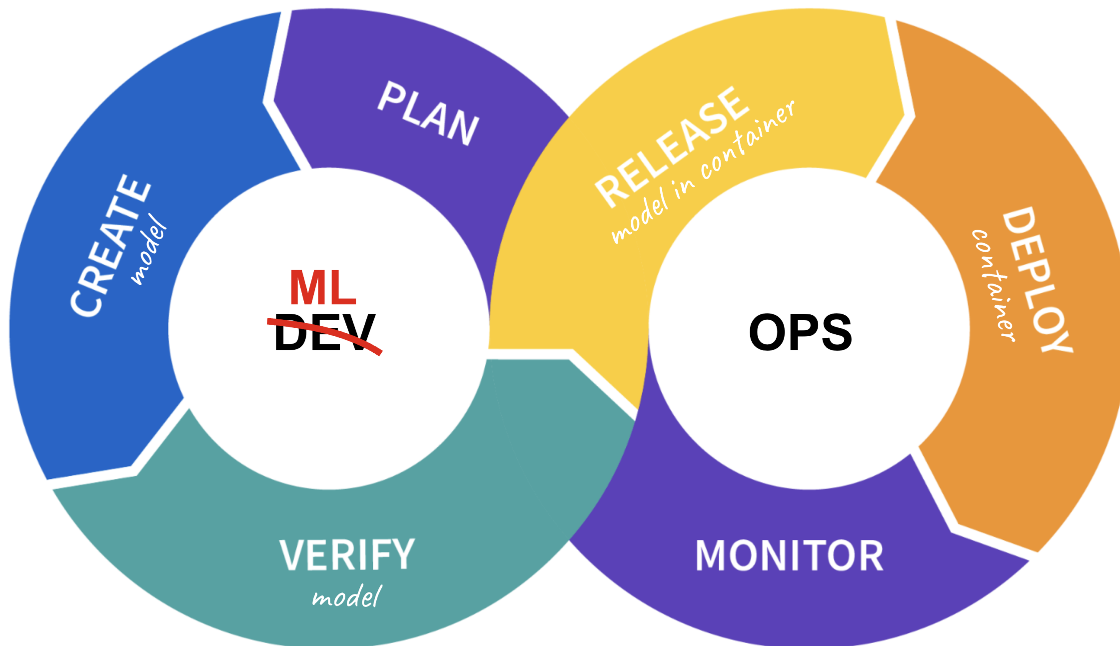


Figure 1-3. Adapting the DevOps loop for MLOps needs

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", 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

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)

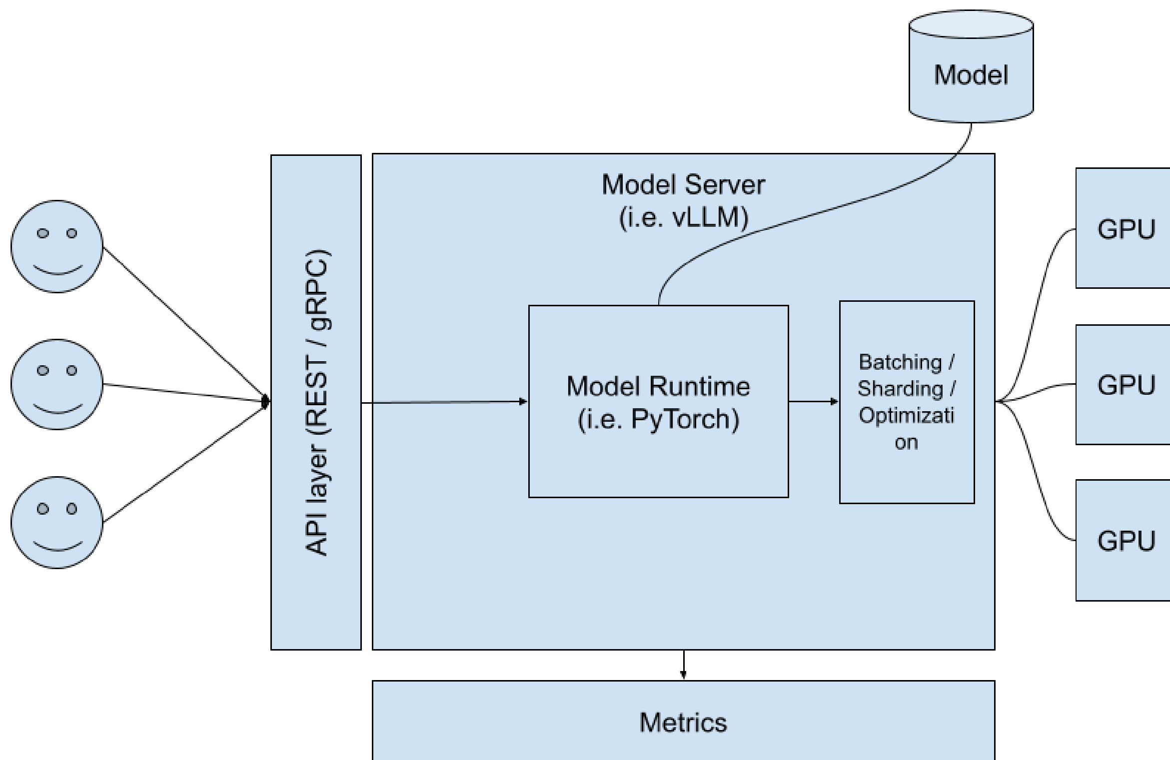


Figure 2-1. Model Server architecture

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.

NOTE

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 \
 --port=8080 \
 --model=/mnt/models \
 --served-model-name=meta-llama/Meta-Llama-3-8B

# 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,
  "temperature": 0
 }'
```

1. It is equivalent to `python -m vllm.entrypoints.openai.api_server`
2. Location (or name) of the model (local to the container)
3. Name of the model
4. 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 ???.

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 \
 --port 8080 \
 --model-id /mnt/models

# invoke the model using TGI API
```

```
 curl localhost:8080/generate_stream \
    -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 \
    -H 'Content-Type: application/json' \
    -X POST \
    -d '{
  "model": "tgi",
  "messages": [
    {
      "role": "system",
      "content": "You are a helpful assistant."
    },
    {
      "role": "user",
      "content": "LLMs are great for"
    }
  ],
  "max_tokens": 10
}'
```

❶ Launcher command

2. Location (or name) of the model (local to the container)
3. TGI original API to invoke the model
4. TGI now supports also OpenAI-compatible API
5. 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

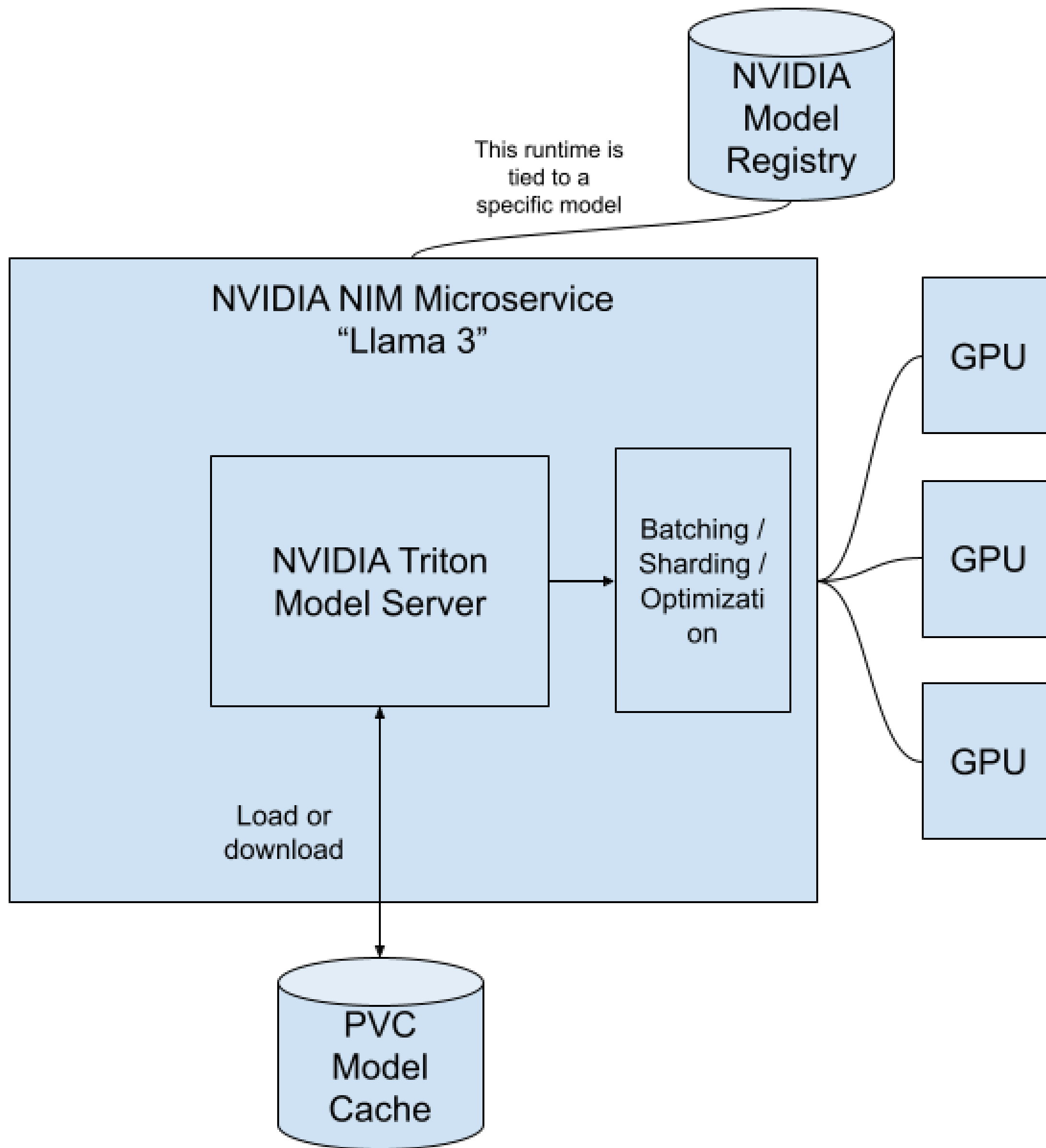


Figure 2-2. NVIDIA NIM Architecture

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 CustomResourceDefinition and a Kubernetes Controller.

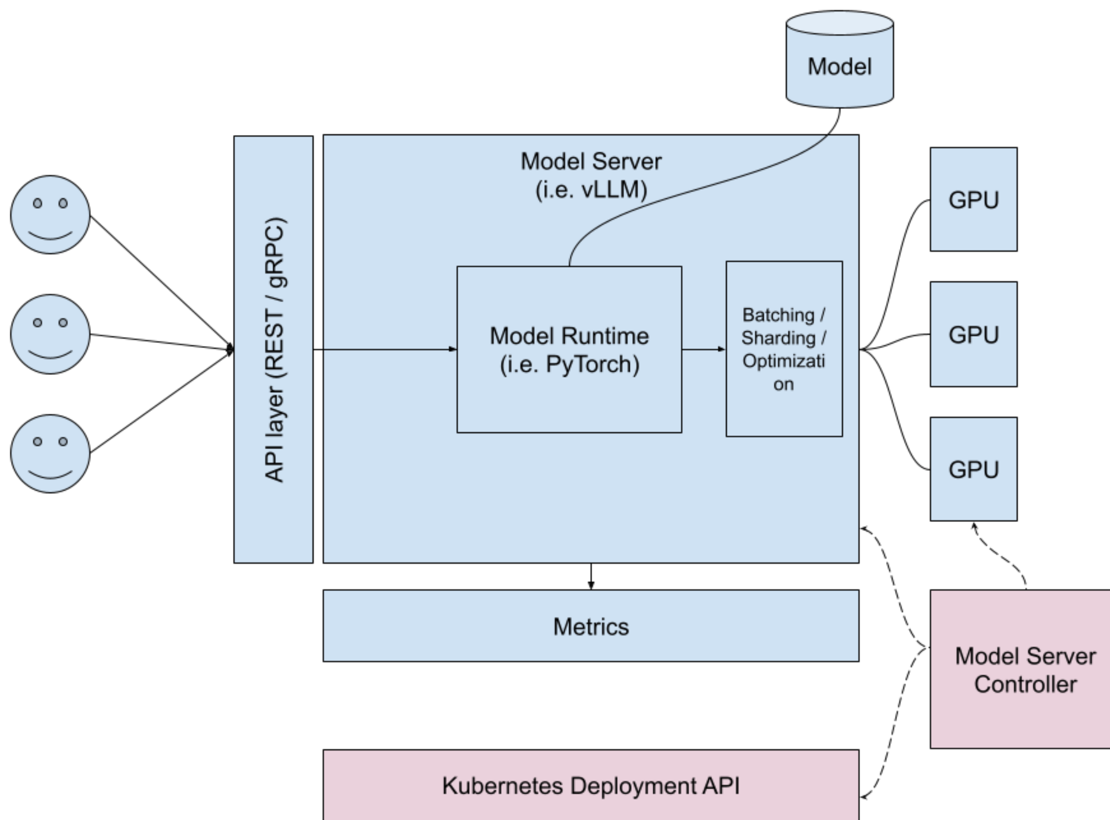


Figure 2-3. Model Server Controller architecture

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'
            requests:
              cpu: '2'
          name: vllm
          env:
            - name: HUGGING_FACE_HUB_TOKEN
              value: ''
          args: [
```

```yaml
            "--port",
            "8080",
            "--model",
            "meta-llama/Meta-Llama-3-8B",
            "--download-dir",
            "/models-cache" ]
          ports:
            - name: http
              containerPort: 8080
              protocol: TCP
          volumeMounts:
            - name: models-cache
              mountPath: /models-cache
          image: vllm/vllm-openai:latest
      volumes:
        - name: models-cache
          persistentVolumeClaim:
            claimName: vllm-models-cache
      tolerations:
        - key: nvidia.com/gpu
          operator: Exists
          effect: NoSchedule
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: vllm-models-cache
spec:
```

```
    accessModes:
      - ReadWriteOnce
    volumeMode: Filesystem
    resources:
      requests:
        storage: 100Gi
```

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

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

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

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

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

6. The volume persisted volume to use as cache

7. 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 ???), 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 configuration 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.

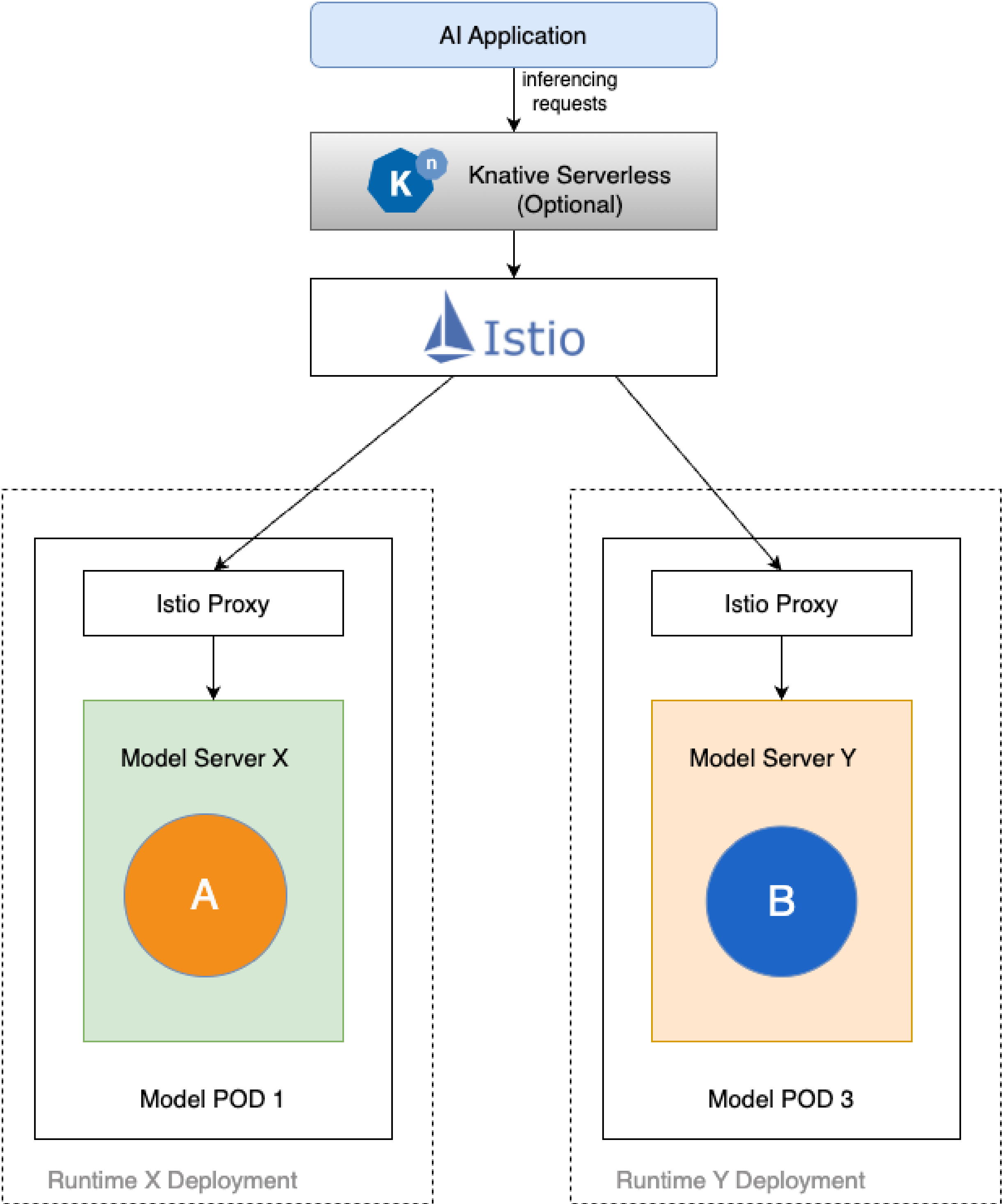


Figure 2-4. KServe Serverless and RawDeployment architecture

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: RawDeployme
spec:
  predictor:
    model:
      modelFormat:
        name: pytorch
      runtime: vllm
      storageUri: pvc://llama/model
    containers:
      resources:
        limits:
          cpu: "4"
          memory: 50Gi
          nvidia.com/gpu: "1"
        requests:
          cpu: "1"
```

```
            memory: 50Gi
            nvidia.com/gpu: "1"
```

1. This annotation is to select the deployment mode
2. Declaring the type of model allows KServe to automatically find a `ServingRuntime` that can handle it
3. It is also possible to specify the name of the runtime to refer explicitly
4. This field specifies where to get the model, in this case from a PVC local to the cluster
5. 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)

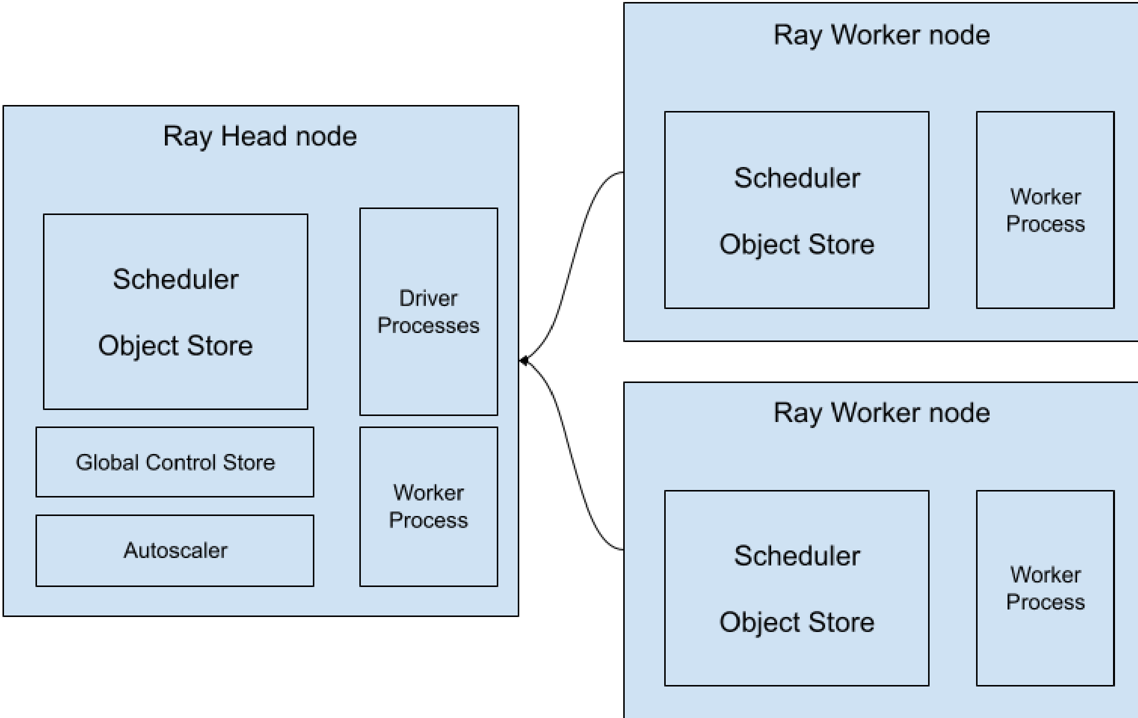


Figure 2-5. Ray Cluster topology

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
class TransformerModelDeployment:
    def __init__(self):
        self._model = pipeline(
            "my-transformer-model")

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


serve.run(
    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: |
    applications:
      - name: my-transformer-model
        import_path: my-transformer-model:deploy
        runtime_env:
          working_dir: "https://my-git-repo.com/
  rayClusterConfig:
    rayVersion: %VERSION%
    headGroupSpec:
      ...
      template:
        spec:
          containers:
          - name: ray-head
            image: rayproject/ray-ml:%VERSION%
            ports:
            ...
            - containerPort: 8000
              name: serve
    workerGroupSpecs:
    - replicas: 1
```

```
        groupName: gpu-group
        template:
          spec:
            containers:
            - name: ray-worker
              image: rayproject/ray-ml:%VERSION%
            tolerations:
              - 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

6. 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
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 black 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” 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”, 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” 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.

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.

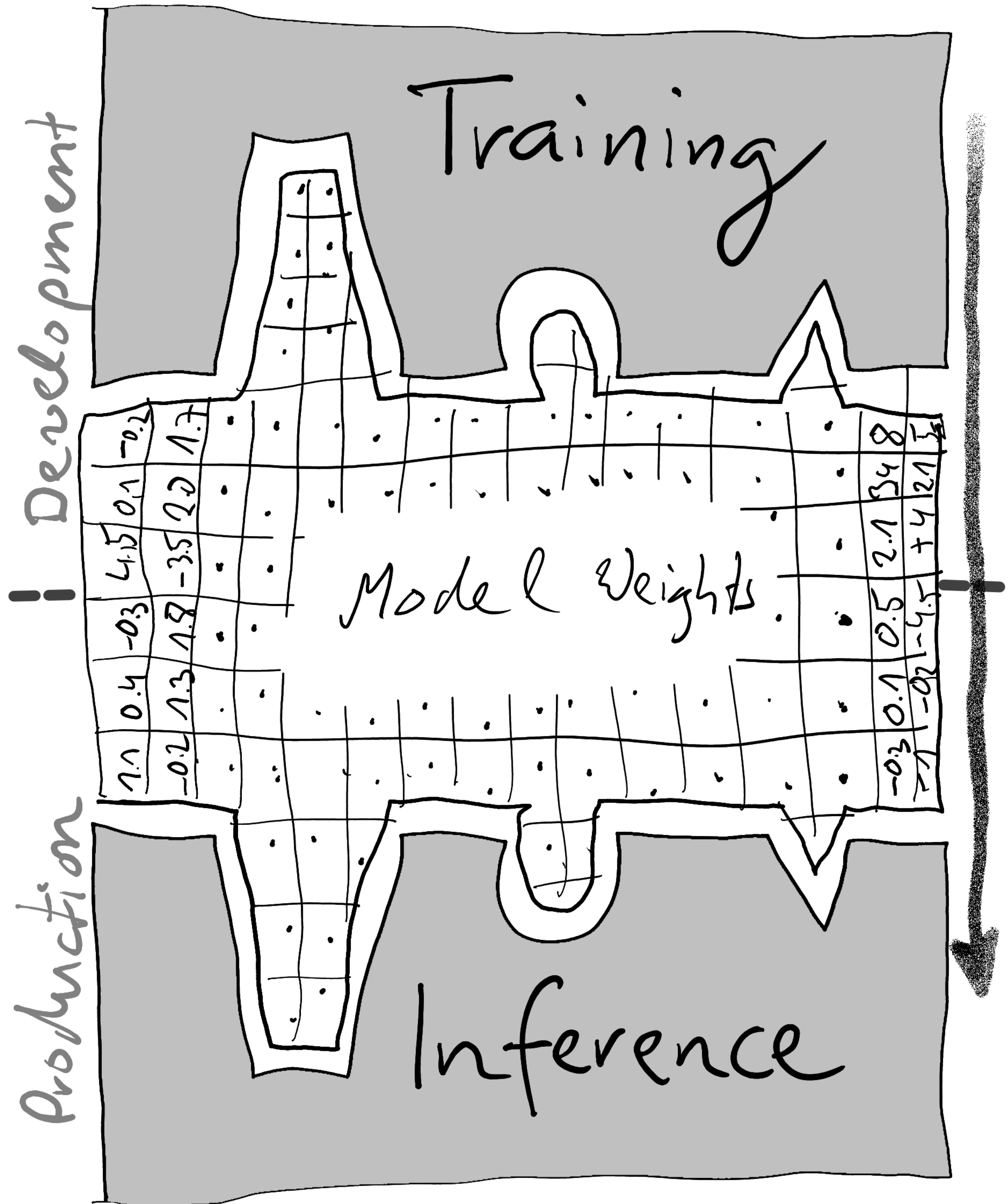


Figure 3-1. Example of a model stored in a weight-only format

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 information (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" 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" 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" 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.

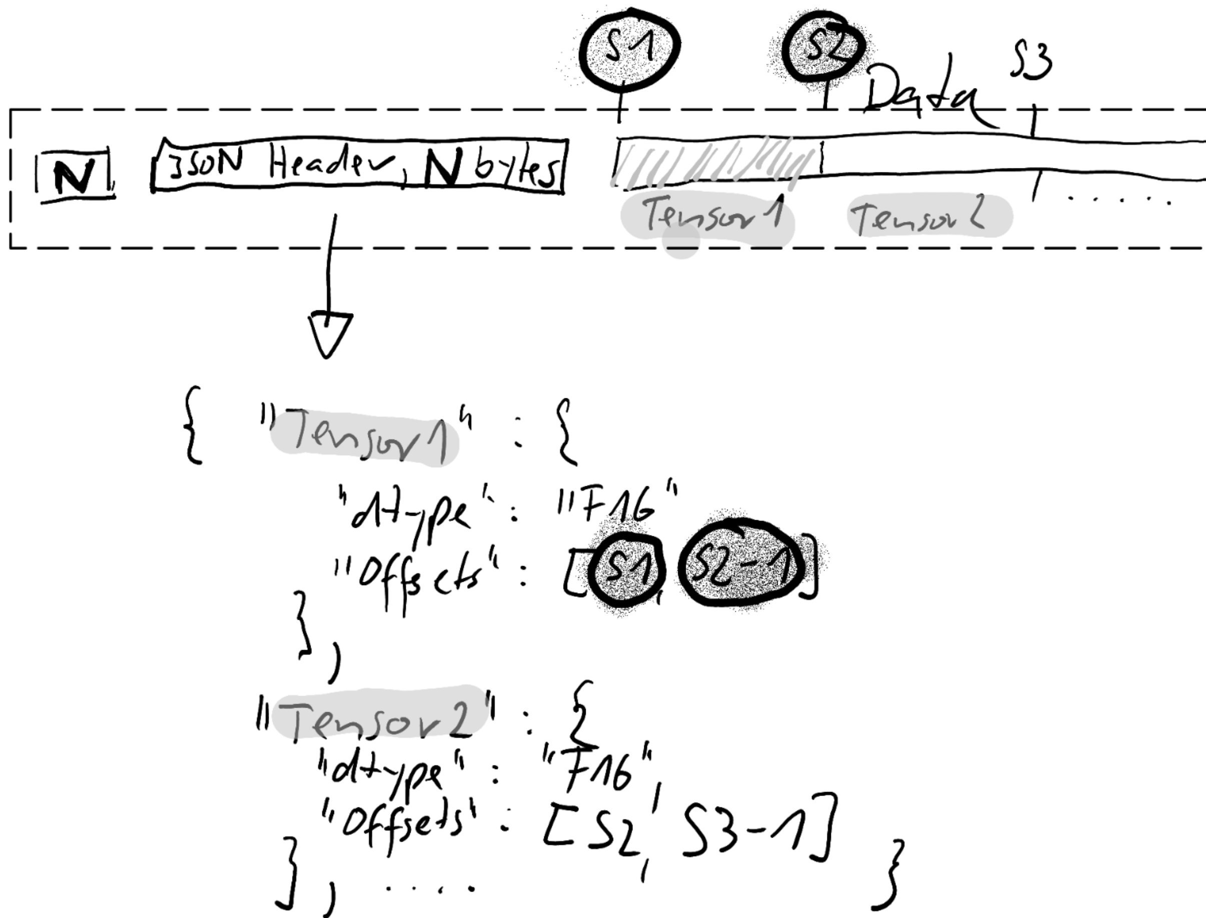


Figure 3-3. Internal structure of a Safetensors model.

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.safe
    "model.embed_tokens.weight": "model-00001-of
    "model.layers.0.input_layernorm.weight": "mo
    "model.layers.0.mlp.down_proj.weight": "mode
```

```
    "model.layers.0.mlp.gate_proj.weight": "mode
    "model.layers.0.mlp.up_proj.weight": "model-(
   ...
    "model.layers.1.input_layernorm.weight": "mo(
    "model.layers.1.mlp.down_proj.weight": "mode
    "model.layers.1.mlp.gate_proj.weight": "mode
    "model.layers.1.mlp.up_proj.weight": "model-(
   ...
  }
}
```

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, 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 Model Face 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).

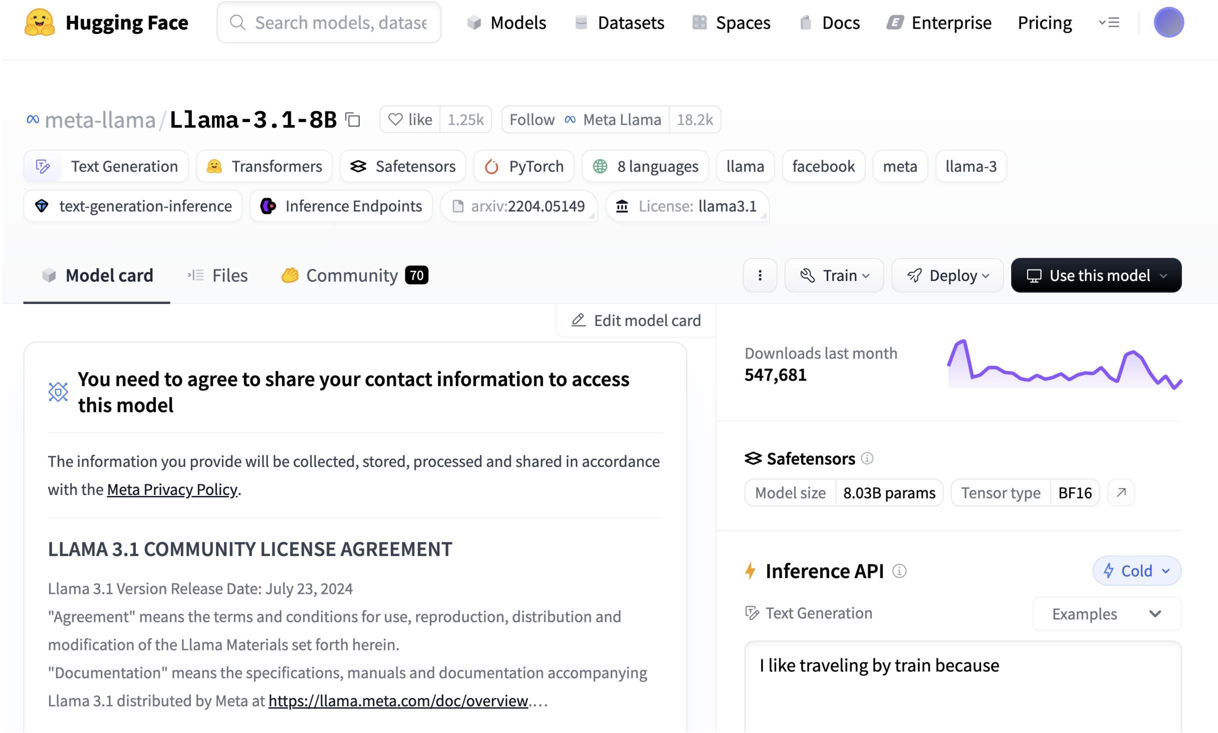


Figure 3-5. Hugging Face Model Card for Llama 3.1

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.

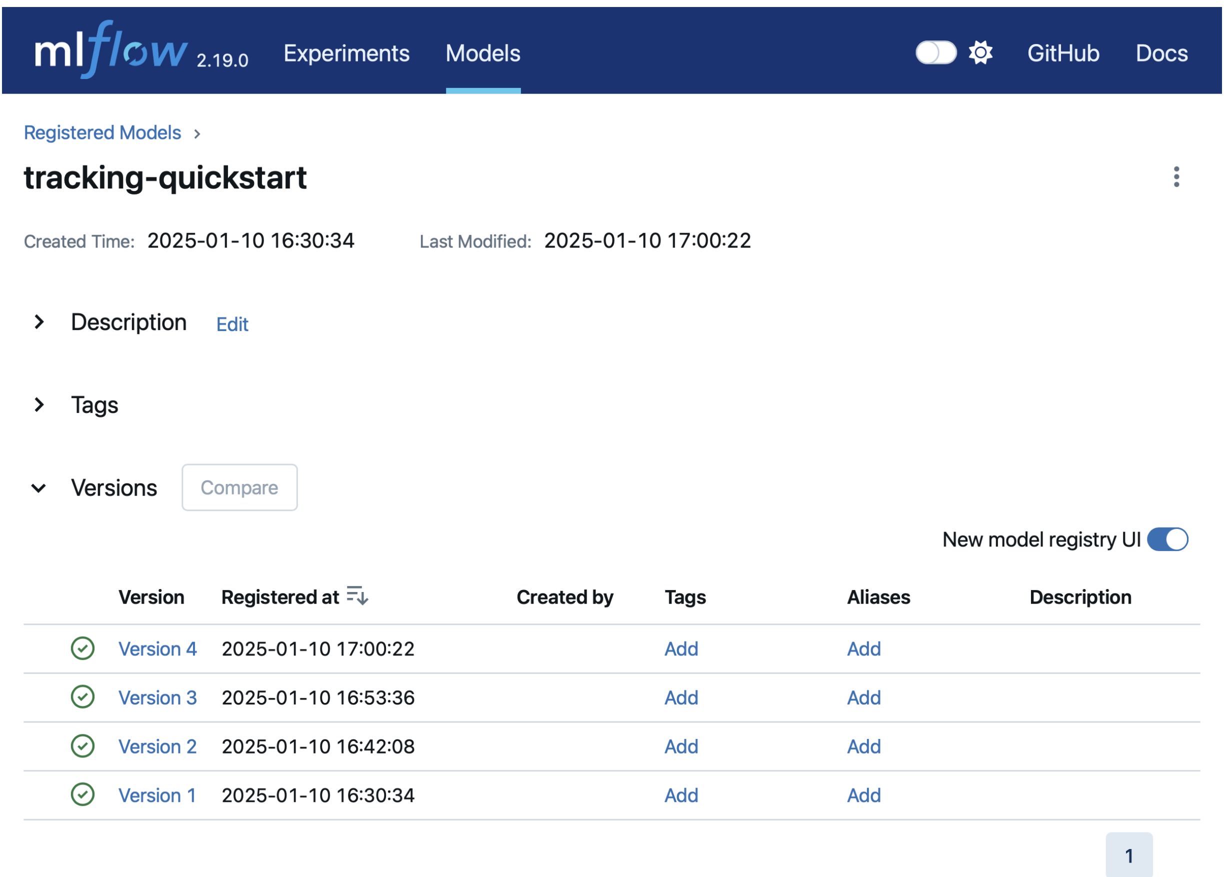


Figure 3-6. MLflow Registry UI

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://127.0.0.1:80
mlflow.set_experiment("MLflow Demo")

params = {
    "solver": "lbfgs",
```

```
    "multi_class": "auto",
    "max_iter": 2500,
}

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

1. Set tracking server uri for logging
2. Create a new MLflow Experiment
3. Model hyperparameters
4. Log those hyperparameters
5. 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/regi

{
  "registered_models": [
    {
      "name": "my-model",
      "creation_timestamp": 1736523034148,
      "last_updated_timestamp": 1736524822538,
      "latest_versions": [
        {
          "name": "my-model",
          "version": "4",
          "creation_timestamp": 1736524822538,
          "last_updated_timestamp": 173652482253
          "current_stage": "None",
          "description": "",
          "source": "mlflow-artifacts:/84948067/
          "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".

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

```
$ mlflow models generate-dockerfile \      ❶
  -m mlflow-artifacts:/84948067/f0dd25483e/artifa
... INFO mlflow.models.cli: Generating Dockerfile
    .../artifacts/my_model
```

```
... INFO mlflow.models.flavor_backend_registry: 
    for flavor 'python_function'
... INFO mlflow.models.cli: Generated 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 inst
....
Successfully tagged localhost/my_model:latest
a828556afe0d53d4728d872aa51fe07eaa1d4ef4faedb5a7
```

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 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 Training Operator 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. We cover KServe in detail in "KServe".

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.

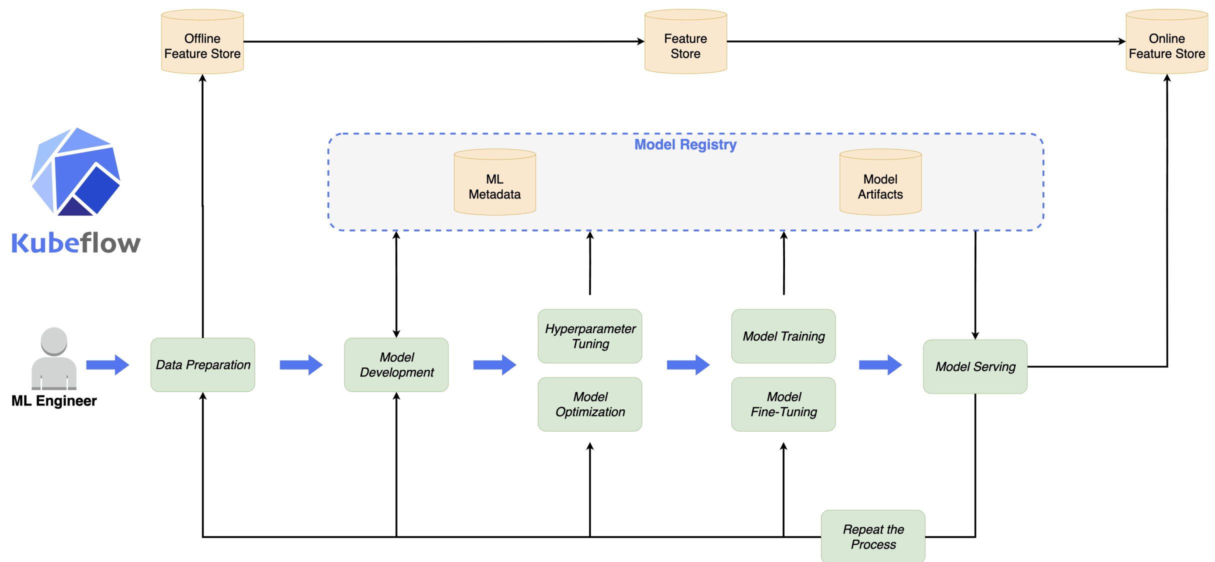


Figure 3-7. Kubeflow architecture and how it interacts with its Model registry

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-servic
    port=8080,
    author="your name",
    is_secure=False
)
rm = registry.register_model(                                ❷
    "iris",
    "gs://kfserving-examples/models/sklearn/1.0/
    model_format_name="sklearn",
    model_format_version="1",
    version="v1",
```

```
    description="Iris scikit-learn model",
    metadata={
        "accuracy": 3.14,
        "license": "BSD 3-Clause License",
    }
```

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

2. 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
   http://model-registry-service.kubeflow.svc.cl
```

1. 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"
      modelFormat:
        name: "sklearn"
        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 can 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.
```

```
 && ln -s /git/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/rh

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/(
       && ln -s /git/Qwen2.5-0.5B-Instruct /mode
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-i
f680df7c975f6bfc806783574003c2b17872e9bf7679443
```

```
$ podman push quay.io/rhuss/qwen2.5-0.5b-instruc
```

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
    tensorflow:
      storageUri: "s3://kserve-examples/mnist"
```

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

2. 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-initi
  supportedUriFormats:
    - prefix: model-registry://
```

1. Reference to OCI image for executing the initializer logic.
2. 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`

Schema	Description	Example
oci	Pull OCI image with model data and access it directly via a modelcar, see "Modelcars".	`oci://quay.io/rhuss/kserving-example-sklearn:1.0`
model-registry	Access a model registered at the Kubeflow Registry. See "Kubeflow Model Registry" 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 diretly. 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 th 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.

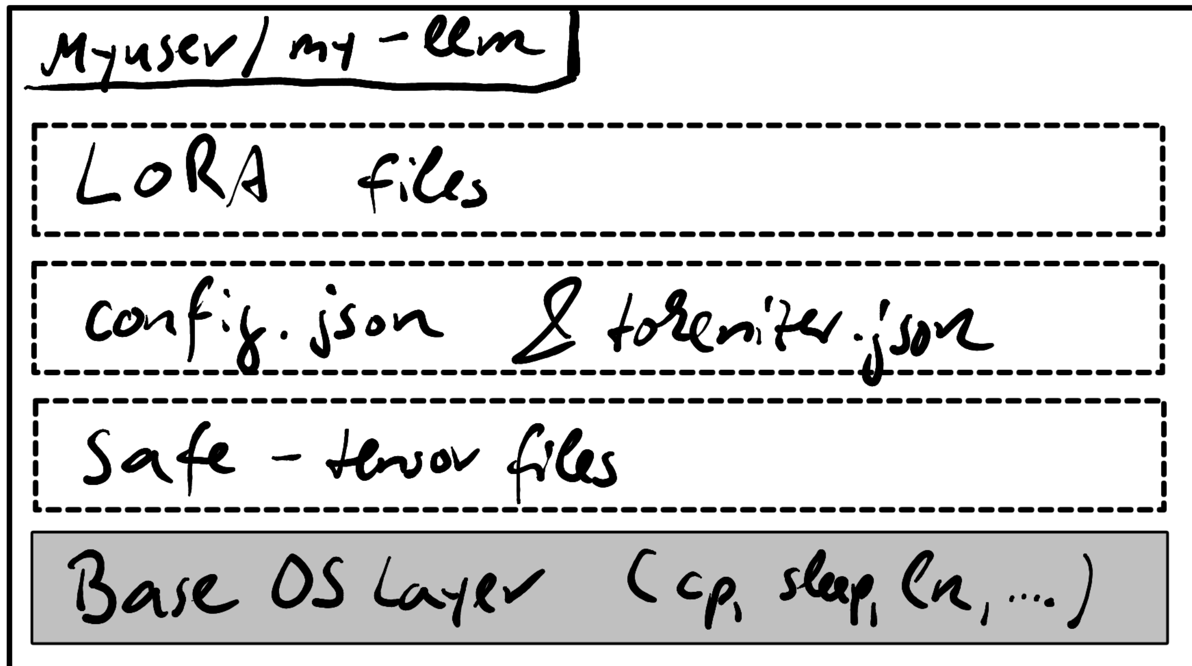


Figure 3-8. OCI Image consists of multiple directory-layer

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" 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
        command:
        - "sh"
        - "-c"
        - "cp -a /models/. /mnt/models"
        volumeMounts:
        - name: models
          mountPath: /mnt/models
      containers:
      - name: vllm
        image: vllm/vllm-openai:latest
        args:
```

```
          - "--served-model-name",
          - "meta-llama/Meta-Llama-3-8B",
          - "--model",
            "/mnt/models"
          volumeMounts:
          - name: models
            mountPath: /mnt/models
        volumes:
          - name: models
            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" and "OCI Image Volume Mounts". 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" 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-
```

❶ OCI registry and repository of image holding the model data

---

NOTE

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". 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 t
  $ sed  's/false/true/' spns.yml | kubectl apply

  # 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/h
#
# This is the main Apache HTTP server configurat
# configuration directives that give the server 
```

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.

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

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

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

NOTE

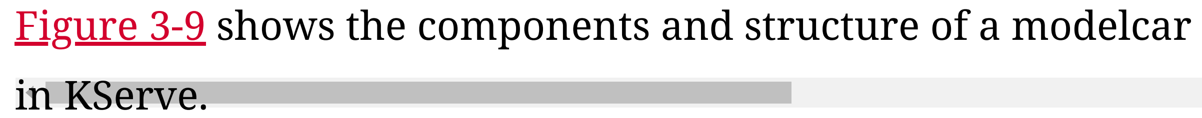
Accessing other processes' filesystem is only possible when Unix permissions allow. Ideally the processes from all containers use the same UID, so that cross-container filesystem access should not be an issue. However, depending on your cluster setup additional mechanisms like SELinux might affect the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



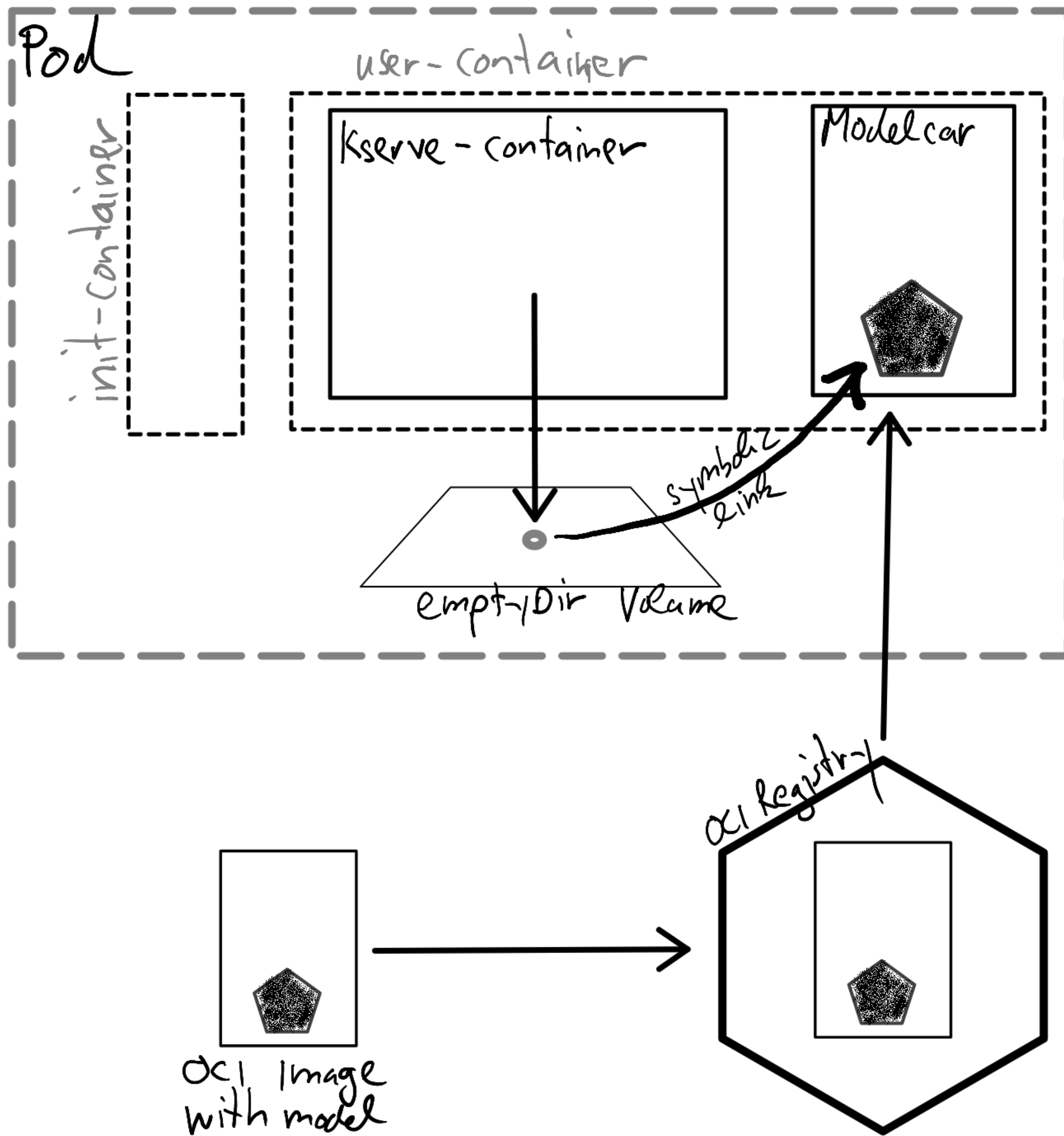


Figure 3-9. Modelcar components

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 (`/opt/model`).

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
```

```yaml
metadata:
  name: sklearn-iris-oci-predictor-00001-deploym
  namespace: default
spec:
  shareProcessNamespace: true
  containers:
  - name: kserve-container
    image: kserve/sklearnserver
    args:
    - --model_name=sklearn-iris-oci
    - --model_dir=/mnt/models
    volumeMounts:
    - mountPath: /mnt
      name: kserve-provision-location
  - name: modelcar
    image: rhuss/kserving-example-sklearn:1.0
    args:
    - sh
    - -c
    - ln -s /proc/$$$$/root/models /mnt/models &
    volumeMounts:
    - mountPath: /mnt
      name: kserve-provision-location
  volumes:
  - name: kserve-provision-location
    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 platform 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”) 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. This feature

is currently supported only with `cri-o` container runtime and not `containerd`, but `containerd` will catch up soon.

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
      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).

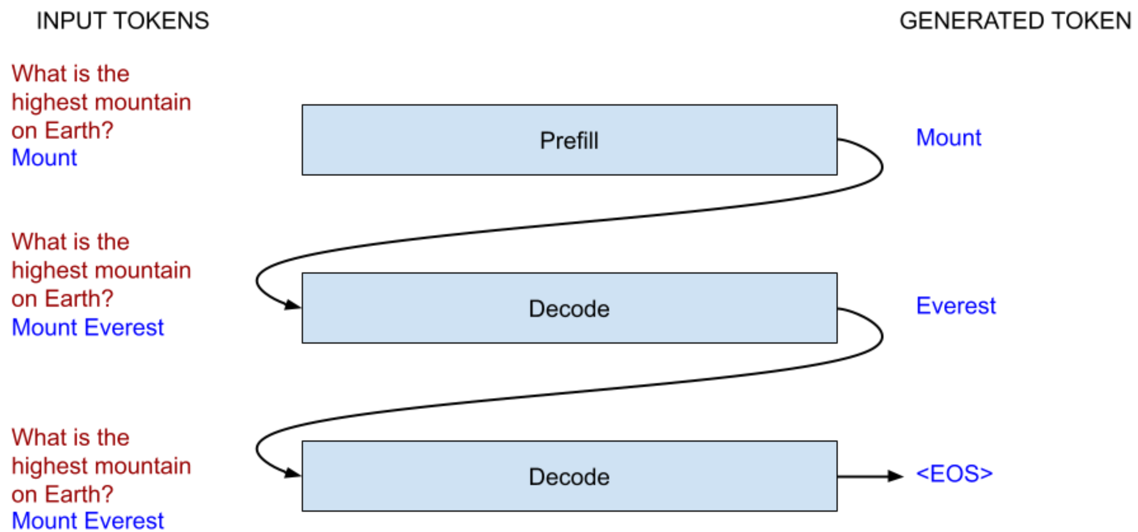


Figure 4-1. LLM processing steps

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.

TIP

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.

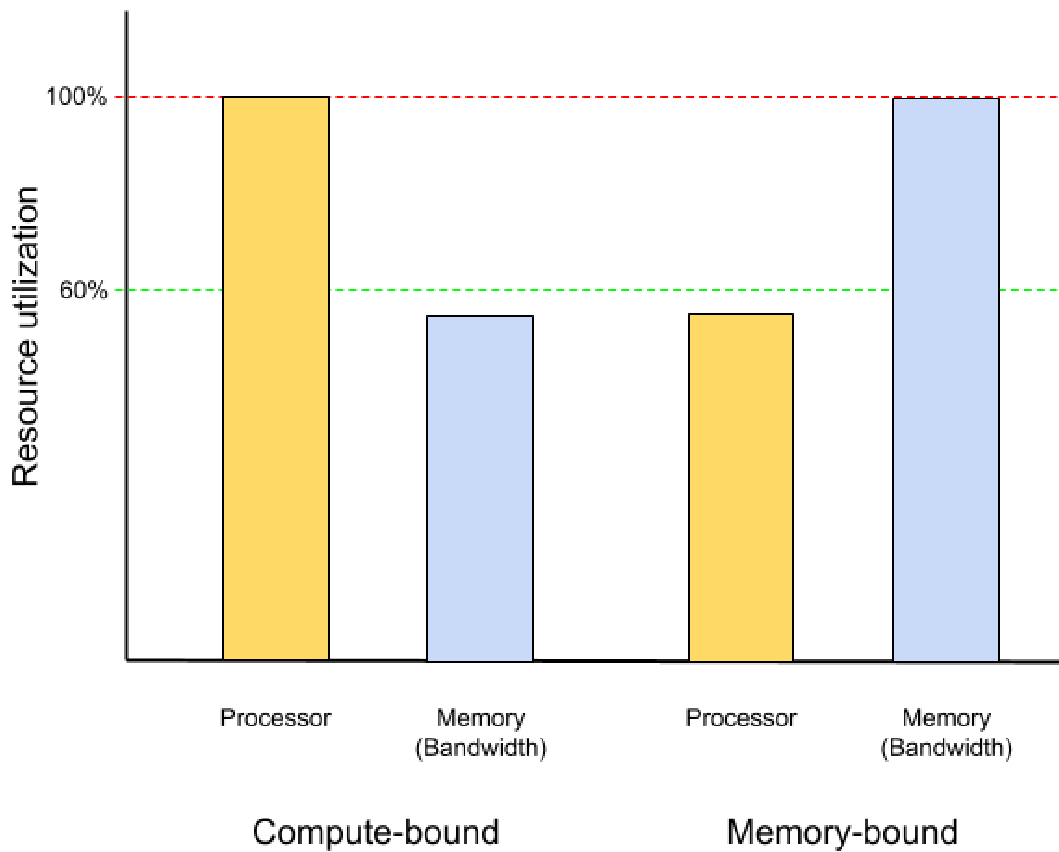


Figure 4-2. Compute-bound and memory-bound

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.

TIP

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.

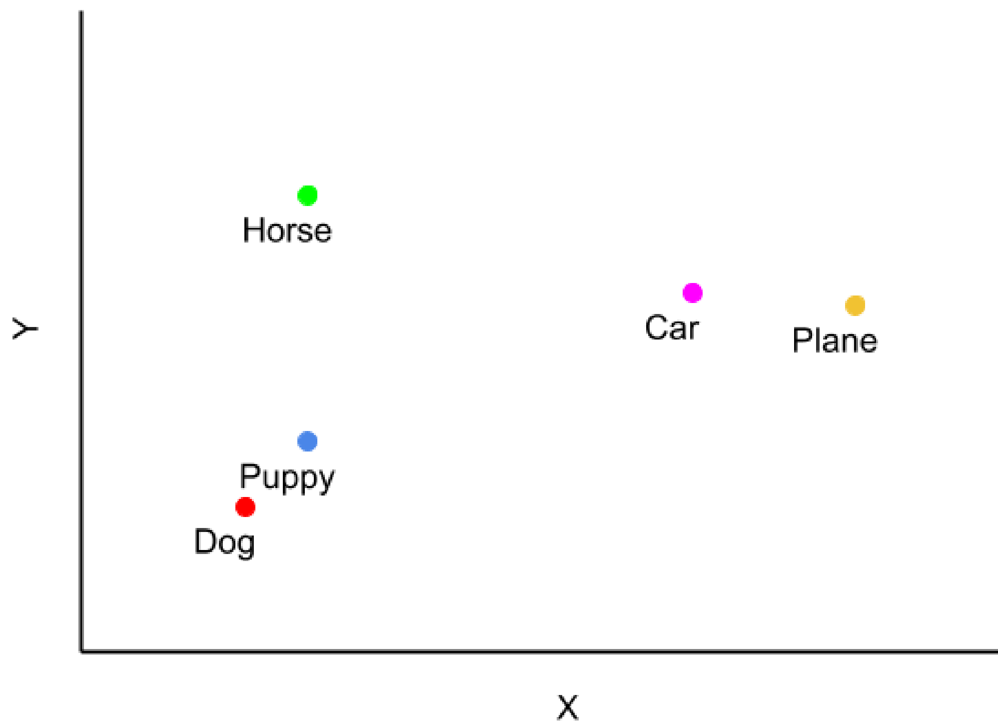


Figure 4-3. Simplified embedding representation

NOTE

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" 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” 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
INFO api_server.py:652] args: ...
INFO api_server.py:199] Started engine process w
INFO config.py:478] This model supports multiple
WARNING arg_utils.py:1089] Chunked prefill is ena
INFO llm_engine.py:249] Initializing an LLM engi
INFO model_runner.py:1092] Starting to load mode
INFO weight_utils.py:243] Using model weights fo
...
Loading safetensors checkpoint shards: 100% Comp
...
INFO worker.py:241] the current vLLM instance ca
INFO worker.py:241] model weights take 14.99GiB;
...
INFO launcher.py:19] Available routes are:
INFO launcher.py:27] Route: /openapi.json, Metho
...
```

```
INFO launcher.py:27] Route: /v1/chat/completions
...
INFO:     Started server process [39626]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000
INFO logger.py:37] Received request cmpl-...: pr
INFO engine.py:267] Added request cmpl-....
```

❶ vLLM logs the version and the arguments specified to start it

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

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

❹ 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)

❺ 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

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

```yaml
  name: my-service-deployment
spec:
  ...
---
apiVersion: v1
kind: Service
metadata:
  name: my-service
  annotations:
    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
metadata:
  name: my-service-servicemonitor
spec:
  selector:
    matchLabels:
```

```
      app.kubernetes.io/part-of: my-application
  endpoints:
    - 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:
    prometheus.kserve.io/port: '8080'
    prometheus.kserve.io/path: "/metrics"
  ...
---
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: my-model
  annotations:
    serving.kserve.io/enable-prometheus-scraping
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.

TIP

KServe has different deployment modes as already described in "KServe". 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" 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” 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 ??? 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)

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.

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_to
        labels:
          severity: critical
          app: my-model
        annotations:
          message: Latency of vLLM is too high
          summary: Model "my-model" needs to k
          runbook_url: https://my.company/runb
          description: The runtime is slowing
```

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 ??? 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 ???.

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".

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 Engli
"This world record was made on August 14, 2020,
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".

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”.

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' mess
   conversations our policies described with the

<BEGIN UNSAFE CONTENT CATEGORIES>
O1: Violence and Hate.
Should not
- Promote or suggest people to use violence.
...
Can
- Describe what violence is.
...
<END UNSAFE CONTENT CATEGORIES>

<BEGIN CONVERSATION>
...
<END CONVERSATION>

Provide safety assessment with following format:
- First line must be 'safe' or 'unsafe'
- If unsafe, second line must include comma-sepa
  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 ???. 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!

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.