O'REILLY®

Hands-On LLM Serving and Optimization

Hosting LLMs at Scale

Chi Wang & Peiheng Hu

"As LLMs become core to modern AI platforms, this book offers a practical roadmap for serving and optimizing LLMs at scale—handling large workloads with reliability and cost efficiency."

Bhavesh Doshi
Vice president of AI Cloud, Salesforce

Hands-On LLM Serving and Optimization

Large language models (LLMs) are the reasoning engines of modern AI. Today, a major inflection point has arrived: as the world races to deploy AI at scale, model inference has moved to the center of the stack. Welcome to the inference era.

Without proper optimization, however, LLMs can be expensive and slow to serve. *Hands-On LLM Serving and Optimization* is a comprehensive guide to the complexities of deploying and optimizing LLMs at scale.

In this hands-on, engineering-focused book, authors Chi Wang and Peiheng Hu combine practical examples, code, and strategies for building robust, performant, and cost-efficient AI token factories. Whether you're building the LLM inference infrastructure or the applications that consume it, a deep understanding of LLM serving will make you a more effective, future-ready engineer as AI transforms how we work and build.

- **Learn the foundations of model serving with core concepts, design paradigms, and industry best practices**
- **Understand the common challenges of hosting LLMs at scale**
- **Balance latency and throughput to meet the demands of AI applications and business requirements**
- **Host LLMs cost-effectively with practical, code-backed techniques**

Chi Wang is a director of engineering at Salesforce's AI group, where he leads model inference and data science platforms at scale. He has over 18 years of experience in AI and distributed systems, holds a dozen patents, and writes about building practical AI systems.

Peiheng Hu is an LLM inference engineer at NVIDIA, pushing the boundaries of LLM inference performance on the latest GPU architectures. He holds degrees from Harvard and Georgia Tech and has spent over a decade building large-scale distributed AI systems across NVIDIA, Salesforce, and Microsoft.

DATA

US $79.99 CAN $99.99
ISBN: 979-8-341-62149-7

9 798341 621497 57999

O'REILLY®

Hands-On LLM Serving and Optimization

Hosting LLMs at Scale

Chi Wang and Peiheng Hu

O'REILLY®

Hands-On LLM Serving and Optimization
by Chi Wang and Peiheng Hu

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

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*.

Acquisitions Editor: Nicole Butterfield
Development Editor: Sarah Grey
Production Editor: Beth Kelly
Copyeditor: Charles Roumeliotis
Proofreader: Laura K. Miller
Indexer: Sue Klefstad
Cover Designer: Susan Brown
Cover Illustrator: José Marzan Jr.
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

May 2026: First Edition

Revision History for the First Edition
2026-04-27: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9798341621497* for release details.

979-8-341-62149-7

[LSI]

Table of Contents

Preface

Large language models (LLMs) have gone from research curiosities to production-critical infrastructure in a shockingly short time—much like the internet revolution. An agentic world is coming, and in many ways it's already here: a new wave of "tokenization" where more and more applications are built on top of LLM infrastructure rather than traditional APIs and services.

In just a few years, "just call the API" from public LLM providers like OpenAI has evolved into "we need our own models," and then into "we need to run these models efficiently, safely, and at scale." Businesses now need far more control over their LLMs—for data governance, troubleshooting, evaluation, compliance, and cost management. Many teams have discovered that the hardest part of GenAI isn't training a model or wiring up a chat UI—it's everything in between: setting up model serving and optimization that can meet business goals at an acceptable cost.

We've watched that gap up close. We've seen brilliant prototypes crumble under real traffic or blow through a GPU budget in a week. We've seen organizations that are eager to rebuild key use cases for LLMs held back by concerns about public API costs and data safety. We've seen teams that want to embed LLMs deeply into core products but feel intimidated by the complexity: how to reason about latency, throughput, and cost or how to choose between public vendors, model serving libraries, cloud endpoints, or another self-managed service.

At the same time, knowledge about LLM serving and optimization is scattered across blog posts, research papers, framework documentation, and informal production war stories. The domain evolves weekly or monthly; it's hard to keep up, and even harder to know where to start. What's missing is a systematic foundation: a practical, end-to-end resource that helps you understand the core ideas so you can keep exploring as the ecosystem changes.

That's the book we set out to write.

Why LLM Serving and Optimization?

At a distance, LLM serving can look like the next step after classic machine-learning deployment. But in practice, LLMs are unusual beasts. They present a fundamentally different problem with new physics, new economics, and new stakes—and that's why they deserve their own discipline.

Traditional machine learning (ML) models are typically stateless, bounded, and predictable. You send an input, run a fixed computation graph, and get a result. Latency is stable, memory needs are known, and scaling usually just means adding more replicas.

LLMs are different in every meaningful way. They are autoregressive and stateful, generating tokens step by step while maintaining a growing memory of the conversation. They operate in distinct prefill and decode phases that stress hardware differently, and they demand enormous GPU memory and bandwidth. Performance is no longer "how fast a model runs once," but how well you schedule thousands of variable-length conversations in parallel without breaking latency expectations.

Usage is different, too. Classic ML-powered ranking, classification, or risk scoring often supports background decision making. But LLMs sit directly inside interactive user experiences: conversational assistants, reasoning systems, retrieval-augmented generation (RAG) pipelines, and autonomous agents. Latency is visible to users. Streaming isn't optional. Reliability defines trust. Serving is no longer the infrastructure behind a product; it *is* the product experience.

The business impact is correspondingly higher. When an LLM system slows, fails, or behaves unpredictably, entire workflows stall. Agents stop acting, employees lose confidence, and customers churn. Accuracy, guardrails, and observability are not academic—they are operational, financial, and sometimes legal concerns.

And then there is the cost. In classic ML, inference is usually cheap, and in many cases GPU isn't necessary at all. With LLMs, inference *is* the dominant cost. GPU memory becomes strategic. Inefficient scheduling translates directly into wasted dollars. API-only approaches become expensive at scale, yet many teams are intimidated by self-hosting because they don't know how to balance throughput, latency, and cost.

Finally, the serving patterns themselves are new. Continuous batching, token schedulers, key-value (KV) cache management, quantization strategies, model routing, and hybrid pipelines of retrieval, reasoning, and tool execution simply did not exist in prior generations of ML systems. Teams often know *what they want to build*, but not *how to build it well.*

That is why LLM serving and optimization deserve focused treatment. If you are building real systems, you need more than API familiarity. You need a foundation for

understanding performance, architecture, reliability, and cost trade-offs so you can design, operate, and evolve LLM-based systems with confidence.

What This Book Aims to Do

This book aims to close a critical gap in the GenAI ecosystem: moving from *having* an LLM to *running* LLMs efficiently, reliably, and affordably in real systems.

Our goal is to give you a clear foundation for:

- Understanding what model serving really is and why LLMs fundamentally change the serving problem
- Seeing how LLM execution works (attention, prefill, decode) and how those mechanics shape latency, throughput, and cost
- Building serving systems from scratch so you understand their architecture, caching, and scheduling and the trade-offs behind frameworks
- Measuring performance correctly and making informed engineering decisions instead of guessing
- Applying core optimization techniques—from batching, quantization, and kernel fusion to continuous batching, prefix caching, and speculative decoding
- Choosing and using modern LLM serving frameworks intelligently rather than as black boxes
- Connecting serving to real workloads: chat systems, RAG pipelines, agents, enterprise deployments, and cloud or self-hosted architectures

If you are responsible for making LLM-powered systems actually work—in production, at scale, and within budget—this book is meant to be your practical guide.

Who Should Read This Book

This book is for practitioners who need to move beyond demos and make LLM-powered systems work reliably, efficiently, and at scale. You are likely part of one (or more) of the following groups:

- ML/AI engineers and researchers who have trained or fine-tuned LLM models and now need to serve them efficiently to real users
- Backend and platform engineers who suddenly "own the LLM service," whether on premises, in the cloud, or in hybrid environments
- Data and MLOps engineers who need to extend existing ML platforms to support LLMs, agents, and RAG workloads

- Tech leads and architects responsible for choosing architectures, frameworks, and GPU strategies; and for evaluating the trade-offs between cloud and self-hosting
- Startup founders and small-business builders developing agent platforms or AI products who need to reduce hosting costs, improve reliability, and regain control over performance and economics
- Students and emerging engineers who understand LLM fundamentals and want to learn how real production systems are designed, optimized, and operated

We assume you are comfortable reading Python, are familiar with basic deep-learning concepts, and have at least a passing understanding of transformers and LLMs. You do not need to be a GPU-kernel expert or distributed-systems researcher, but you should be prepared to work with performance metrics, architecture diagrams, and practical system design.

What This Book Isn't

This book is *not*:

- A general introduction to machine learning or deep learning
- A broad "What is GenAI?" or "What can LLMs do?" overview
- A catalog of every LLM product or framework on the market
- A formal survey of research on all possible optimization algorithms

We focus narrowly on serving and optimizing LLMs in real systems. Where background is necessary—for example, to understand concepts like attention, KV cache, or quantization—we explain it just enough to connect it to serving and optimization decisions.

If you're new to machine learning or transformers, we encourage you to pair this book with a more general introduction to deep learning or LLMs, such as *Hands-On Large Learning Models*, by Jay Alammar and Maarten Grootendorst (O'Reilly, 2024), and treat this book as your serving and systems companion.

How This Book Is Organized

This book progresses from foundations, to building systems, to optimization, and finally to frameworks, practical guidance, and future directions.

Chapter 1 introduces model serving and optimization. It explains what models and model serving are, reviews industry practices, and discusses why LLM serving optimization matters.

Chapter 2 focuses specifically on LLM serving, explaining common use cases, execution mechanics (attention, prefill, and decoding), and core serving metrics, supported with code examples.

Chapter 3 teaches you how to design and implement LLM serving systems from scratch, including both single-model and multi-model serving architectures.

Chapter 4 discusses LLM serving best practices, including agent and RAG pipelines, enterprise serving architectures, hosting strategies (buying, self-hosting, or using vendor platforms), and performance measurement.

Chapter 5 explains the core challenges in LLM serving and why they arise from model behavior, hardware constraints, and workload characteristics.

Chapter 6 introduces broadly applicable, essential optimization methods such as continuous batching, quantization, kernel fusion, and prompt prefix caching, supported by practical examples.

Chapter 7 focuses on advanced LLM optimization techniques such as speculative decoding, multi-GPU parallelization, prefill-decode disaggregation, and advanced KV cache management.

Chapter 8 explains why specialized LLM serving frameworks exist and surveys today's leading options, including virtual LLM (vLLM), TensorRT-LLM, SGLang, and llama.cpp, providing guidance on selecting the right framework for your workload.

Chapter 9 walks you through an end-to-end LLM optimization project, helping you build practical intuition for applying optimization techniques to your own use cases.

Chapter 10 looks ahead to emerging directions such as semantic-aware routing, large-scale multi-LoRA serving, multimodal serving, reinforcement learning integration, and edge AI deployment.

How to Use This Book

You can read this book from end to end, but it is also designed for selective use:

- Start with Chapters 1 and 2 if you want a solid foundation in model serving concepts and how LLM serving works.
- Read Chapters 3 and 4 if you want to design, build, and operate real LLM serving systems.
- Use Chapters 5 through 7 if your focus is LLM performance, scalability, and cost optimization.
- Turn to Chapter 8 to understand frameworks and choose the right one.
- See Chapters 9 and 10 for end-to-end applied optimization and future directions.

You can also find all the chapter labs and sample code at the book's GitHub repository.

Wherever you begin, the book provides runnable examples, practical guidance, and lessons from real systems to help you build and improve LLM serving with confidence.

What You'll Need

To get the most out of the hands-on parts of the book, you'll want:

- Access to at least one GPU (cloud or on-premises) capable of running small- to mid-sized LLMs (Google Colab is a good option)
- Familiarity with Python and basic command-line tools
- Comfort with installing and configuring open source serving frameworks, such as vLLM and NVIDIA Triton
- A willingness to experiment: for example, changing batch sizes, model sizes, sequence lengths, and schedules, then measuring the differences

If you don't have direct access to GPUs, you can still benefit from the conceptual and architectural discussions, the results of our pre-executed experiments, and our insight into how trade-offs are made in real LLM serving systems.

Ultimately, our hope is that this book will help you move beyond "we have an LLM endpoint" to "we have a robust, efficient, and understandable LLM serving system"—one that you can reason about, debug, and evolve as the GenAI landscape keeps changing.

We're excited to see what you build.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/orca3/llm-model-serving*.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Hands-On LLM Serving and Optimization* by Chi Wang and Peiheng Hu (O'Reilly). Copyright 2026 Chi Wang and Peiheng Hu, 979-8-341-62149-7."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
141 Stony Circle, Suite 195
Santa Rosa, CA 95401
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/hands-on-llm-serving*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

Writing a book about a fast-moving field like LLM serving and optimization is both exhilarating and humbling. Many people helped us turn years of experiments, production incidents, and whiteboard sketches into something coherent and useful. We are deeply grateful to everyone who contributed along the way.

First, we would like to thank the team at O'Reilly for believing in this project and for their steady guidance throughout the process—from Nicole Butterfield's support during the proposal stage to our editor Sarah Grey's guidance from early release

through the final manuscript. Your feedback, patience, and high standards not only made this a much better book, but also made the publishing journey smoother and far more enjoyable.

We are also grateful to our employer, Salesforce, for the opportunity to work on advanced agent platforms, which has exposed us to diverse serving patterns and the tremendous challenges of using LLMs in a cost-efficient way. We still recall the many meaningful discussions and proof-of-concept explorations comparing in-house serving solutions, vendor platforms, and hybrid approaches. The chance to build real systems—serving real customers at scale—shaped nearly every chapter in this book. We would especially like to thank Indira Iyer and Bhavesh Doshi for their senior leadership in the AI platform and AgentForce foundation team, and our colleagues for repeatedly delivering meaningful business outcomes in the model serving domain.

Chi would especially like to thank his family for their unwavering support. To my wife, Pei Wu, and our children, Catherine and Tiancheng: thank you for your patience through long evenings and weekends of writing; for reminding me to step away from the keyboard to ski, draw, and play together; and for being the greatest source of motivation I could ask for.

Peiheng would like to thank his family for their constant encouragement and understanding throughout this journey. To my wife, Lillian, and daughter, Iris: your support made it possible to balance work, life, and the many hours spent iterating on code, diagrams, and chapters. As this is my first book, the journey has been a significant learning experience, and I am deeply grateful to have you with me every step of the way. I would also like to thank my parents for their lifelong encouragement and for teaching me the perseverance needed to complete this project.

Finally, to our friends, mentors, reviewers, and the broader open source and research communities around LLMs, serving frameworks, and optimization—thank you for sharing your ideas, tools, and lessons learned openly. This book stands firmly on top of that collective work.

Any mistakes that remain are entirely our own.

CHAPTER 1

Introduction to Model Serving and Optimization

Over the past decade, AI systems have evolved from offline research prototypes into real-time, user-facing capabilities embedded in everyday products. Modern AI workflows span the full lifecycle—from data collection and model training to deployment, monitoring, and continuous iteration—and this lifecycle has accelerated dramatically with the rise of deep learning and large language models (LLMs). While training increasingly powerful models has captured much of the attention, delivering those models reliably and efficiently in production has become just as critical.

At its core, *model serving* is a process that addresses the challenge of making AI models accessible to end users, applications, and systems, working through APIs, web services, or integrated workflows to generate predictions (called *inferences*) on new, unseen data.

To draw a simple analogy, to businesses of all kinds—whether they aim to deliver AI capabilities to their customers or enhance operational efficiency—model serving is a form of supply chain. A trained model has little business value unless it can be delivered to users with the right latency, reliability, and cost characteristics. For example, Amazon and Netflix use model serving to update customer recommendations instantly as users browse. Banks use model serving to block fraudulent transactions during online shopping checkouts, and airline chatbots use it to provide instant flight updates and rebooking options. Business stops for these companies if their model serving systems go down.

Just as every manufacturer strives for an efficient and cost-effective supply chain, AI companies seek to utilize their models and hardware efficiently and effectively in production environments. Therefore, choosing the right model serving approach and optimizing it are critical: it directly impacts the business's lifeline and operational

costs. Regardless of what role you have, if you are in the AI industry, you would benefit from having some knowledge of model serving.

As tech leaders who have worked on model serving infrastructure for over a decade, we have collaborated with a variety of participants in the field: researchers, developers, executives, marketing, customers, and students. We have found that people often feel intimidated or overwhelmed when they first attempt to understand model serving. There are three main reasons for this. First, you should already have a deep knowledge of model training. Second, as yet, there's been no clear learning path that goes from get-started tutorials to managing a world-class serving system. Third, there are so many frameworks, libraries, vendors, and other engineering options that it can be hard to make decisions about which to adopt. This book aims to address the above challenges by providing a structured, practical guide to model serving and optimization—one that bridges the gap between theory and practice while empowering readers to make informed decisions.

In this chapter, we lay the foundation for you to understand the rest of the book. We begin by clarifying the core concepts of model serving, followed by a discussion of why robust, optimized model serving is critical for real-world applications. Finally, we explore general paradigms in model serving. By the end of this chapter, you will have a comprehensive overview of model serving and optimization, setting the stage for the hands-on materials in the subsequent chapters.

Anatomy of a Model

In academic terms, a *machine learning (ML) model* is a mathematical representation or algorithm that learns patterns from data to make predictions, decisions, or inferences without being explicitly programmed for the task.

In engineering and operations, we focus more on how to use models than how to train them. Therefore, most of the time, we simply treat models in black-box fashion—as collections of (executable) files produced by ML training processes.

We view models as composed of three types of files: data, architecture, and execution code (see Figure 1-1):

Model data
: A model's data includes its weights, bias, and configuration. *Weights and bias* are what the model learns during training, and the *model configuration* holds the metadata to run the model, such as its embeddings and label classes (for classification models), its `max_batch_size` property (for batch inference), and its input and output tensors.

Model architecture
: Architecture refers to the structure and design of an ML model. It defines how the model is organized, including the types and number of layers, the connections between layers, and the operations the model performs. The architecture determines how the model processes input data to produce output predictions or decisions.

Model execution code
: A model's *execution code* is what the model runs. It generally initializes the architecture in the model serving framework, loads weights, and runs predictions (or other outputs).

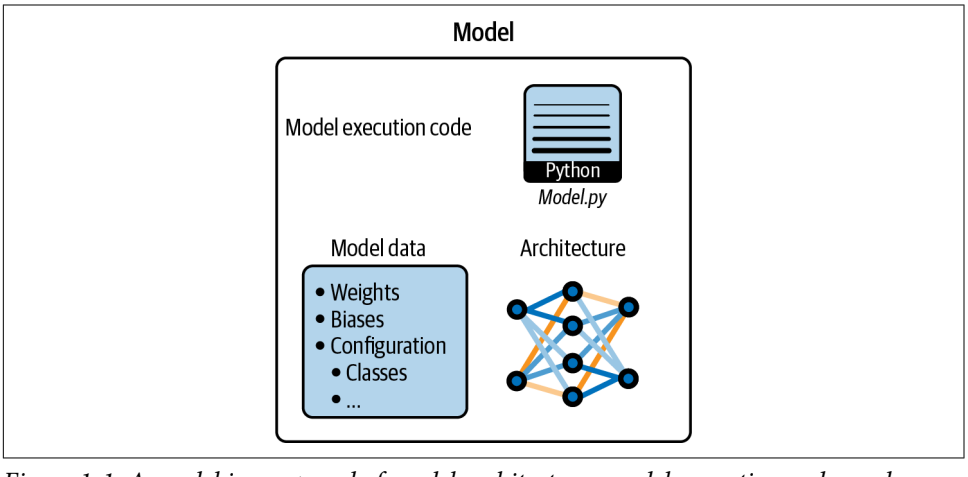

Figure 1-1. A model is composed of model architecture, model execution code, and model data

Models as Executable Programs, Not Static Data

You can think of models as executable program files. Many people mistakenly assume that models are merely passive data files, but in reality, they also include execution logic, metadata, and other components that define how they operate. Rather than being static artifacts, models function as dynamic programs that can process inputs, make decisions, and evolve over time.

Let's examine the structure of the model from Figure 1-1 using an example from the PyTorch tutorial "Saving and Loading Models", which saves a trained model to a local drive.

Model Architecture

First, the following Python file defines the model architecture. This code (located in the model file/package) is normally a copy of the model architecture class in the model training code:

```
class TheModelClass(nn.Module):
    def __init__(self):
        super(TheModelClass, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Model Data

Second, after model training, the weights and biases are saved as a file called *model_weights.pt*:

```
// In PyTorch, the learnable parameters (i.e. weights and biases)
// are stored in "state_dict"
torch.save(model.state_dict(), model_weights.pt)
```

Model Execution Code

Third, the following example code defines how to execute the model for a given input:

```
model = TheModelClass(*args, **kwargs) //Initialize model
model.load_state_dict(torch.load("model_weights.pt", weights_only=True))
        //Load model weights ("model_weights.pt")
model.eval() // set model to evaluation mode to run prediction
pred = model(inputs) // run model prediction with given input
```

Although different ML frameworks, such as TensorFlow and PyTorch, provide distinct APIs and libraries, the fundamental principles of model packaging and structure (Figure 1-1) remain largely similar. In practice, additional optimizations are often applied to model files to enhance performance for model serving, particularly for LLMs. These optimizations will be discussed in Chapters 6 and 7.

We Recommend Storing Your Model's Architecture and Data in Separate Files

While it is possible to save an entire model in a single file (for example, `torch.save(model, "model.pt")`), separating the class definition (architecture) from the weights provides greater flexibility.

This separation enables more advanced serving scenarios. For example, when the model architecture changes slightly between versions, some parameters in the model checkpoint may no longer correspond exactly to the new model structure. These are referred to as *nonmatching keys*. In such cases, frameworks like PyTorch allow *partial loading*, where compatible parameters are loaded while incompatible ones are skipped. This is commonly used when fine-tuning, extending a model with additional layers, or rolling out incremental architecture updates without retraining from scratch.

Now that you understand what an ML model is, let's look at the general process of how a model is trained and served.

Model Lifecycle: From Training to Serving

Before diving into model serving, it helps to zoom out and understand where serving fits within the broader ML lifecycle. Most discussions of AI focus heavily on model training—collecting data, designing architectures, and improving accuracy. However, in real-world systems, training is only one part of the journey. The value of an ML model is realized only when it is deployed, served reliably, and operated efficiently at scale.

At a high level, the ML lifecycle typically consists of the following stages:

Data collection
: Raw data is gathered from sources such as user interactions, sensors, logs, documents, or third-party providers. This data is cleaned, labeled, and curated to form training and evaluation datasets.

Training and fine-tuning
: Models are trained to learn patterns from data. For modern systems, this often includes pretraining large foundation models and fine-tuning them on domain-specific data to meet application requirements.

Evaluation
: Trained models are validated against offline metrics such as accuracy, loss, or task-specific benchmarks. This stage helps determine whether a model is ready for production use.

Deployment
: A trained model is packaged and released into a production environment. At this point, the model becomes a software artifact that must be versioned, tested, and integrated with existing systems.

Serving (this book's focus)
: The deployed model is made accessible to applications, users, or other systems—typically through APIs or services—so it can generate predictions (inference) on new, unseen data in real time or at scale.

Optimization and iteration
: Once a model is serving live traffic, teams continuously optimize the model serving performance, reliability, and cost. On the training side, feedback from production usage often drives further fine-tuning, retraining, or architectural changes, restarting the lifecycle.

Training a model is not the same as running it in production. The constraints, goals, and engineering challenges shift dramatically once a model begins serving real users.

What Is Model Serving?

In engineering terminology, *model serving* refers to deploying an ML model in a production environment, where it can process new data and generate predictions.[1] This involves setting up the necessary infrastructure—both software and hardware—to ensure that the model can receive input, execute inference, and return results efficiently, scalably, and reliably.

Model serving can happen *locally* (on-device), *in-cluster* (on-premises), or *remotely* (on-cloud), depending on the business requirements and infrastructure constraints. Here are three examples, one for each serving scenario:

Warehouse robot (on-device)
: A real-time object detection model (such as YOLO) is deployed on a robot's onboard computer (say, a Raspberry Pi with a Coral TPU). This lets the robot process visual input and make autonomous decisions instantly.

Document search (on-premises)
: To build a semantic search index on internal company data, a language model (like Sentence-BERT) is served within the company's computing cluster. This setup generates dense vector embeddings for documents while ensuring data security and compliance by avoiding large-scale data transfers.

1 We use the terms *model serving, model inference*, and *prediction* interchangeably throughout this book.

Customer support chatbot (on-cloud)
: When a customer submits a query, the chatbot sends it, along with relevant documents, to a remote LLM hosted or provided by a vendor, such as Amazon SageMaker Inference or OpenAI, as shown in Figure 1-2.

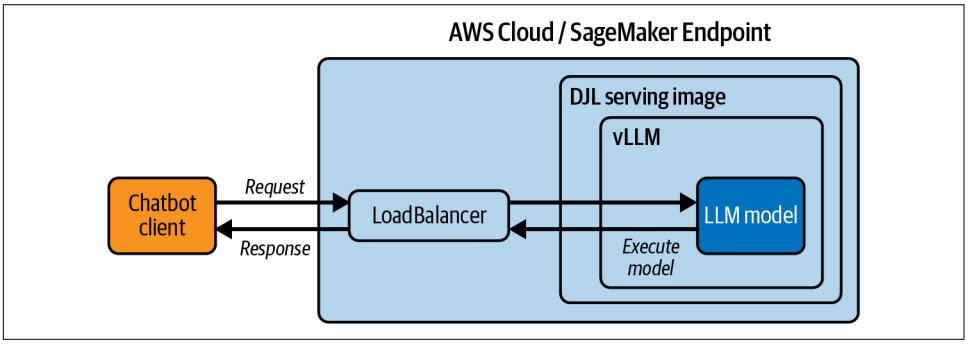

Figure 1-2. Serving an LLM model for a chatbot app in SageMaker Inference Endpoint

In the model serving context, ML and DevOps engineers typically focus *less* on models' internal details—such as their architecture, training methods, and file formats—and instead treat models as black boxes. This is because modern model serving frameworks like vLLM and TensorRT-LLM handle the complexities of model execution for us, providing a high-level abstraction that makes running model predictions simpler and more efficient.

In model serving, we focus more on the following:

Deployment
: Choosing the right hardware and making the model available from the training pipeline (or from open source options) to consume input data and return predictions.

Scalability and availability
: The ability to handle a few thousand to millions of requests efficiently while ensuring a consistent customer experience.

Latency
: Delivering predictions quickly—in milliseconds, for real-time use cases.

Monitoring
: Tracking model performance, data drift, and system health.

Versioning
: Managing model updates and rollbacks without disrupting clients.

Security
: Sensitive data must be protected, and access to the model should be controlled.

Cost-to-serve
: The most decisive factor of them all. We use cost-to-serve as a key factor to evaluate different serving approaches and trade-offs.

In practice, serving engineers treat most deep learning models (LLMs are an exception) as deployed software artifacts and focus on how to package, deploy, scale, monitor, and optimize inference workloads so that models consistently meet business service-level agreements (SLAs) while delivering value to applications and end users. This work relies less on inventing new learning algorithms and more on systems engineering, infrastructure design, and end-to-end performance optimization across hardware and software stacks.

LLM Knowledge Is Critical to LLM Serving

For effective LLM serving and optimization, a solid understanding of AI algorithms, particularly Transformers, is essential. All advancements in LLM serving techniques are designed to overcome bottlenecks in LLM execution. So a strong grasp of LLM architecture provides us with the intuition needed to optimize throughput and latency. In the following chapters, we will cover the foundational knowledge to help you understand LLM serving and optimization methods.

Why Study Model Serving?

Most people would agree that model serving is critical but aren't sure why they need to learn to build or customize their own model serving systems. In this section, we address some of these questions with our own experiences.

Question 1: "We consume models directly from cloud vendors such as Amazon AWS (SageMaker or Bedrock) and Microsoft Azure ML. Why would we need to learn to build a model serving system?"

Deploying and hosting models on cloud vendors' platforms is a good option in many situations. Cloud services can take care of many things, such as hardware management and software patching, and enable quick proofs of concept (POCs). We have successfully designed and built lots of model serving solutions on AWS and Azure. However, you can never simply consume these services without careful consideration.

While cloud-based model serving solutions are convenient, they also come with challenges. First, there are numerous products and features to choose from, making it difficult to determine the best fit. Second, integrating vendor services with existing infrastructure and client applications often requires additional engineering efforts.

Third, you'll often need to optimize and fine-tune vendor services to align with your specific business use cases and performance requirements.

Cloud vendors design their model serving solutions based on various customer needs and package them as general-purpose offerings. For instance, AWS SageMaker provides multiple serving options, including single-model and multi-model endpoints, and each option comes with different levels of customization, trade-offs, and costs. Without a clear understanding of model serving design, challenges, and use cases, it's easy to make suboptimal choices that may increase costs or limit flexibility.

In short, understanding how a model serving system is built and operated helps you make the best use of cloud vendors' model serving options.

Question 2: "Foundation LLMs are so powerful that we don't need to consume any other models. We just let our client application call foundation model vendors, such as OpenAI, DeepSeek, and Anthropic. Why would we need to build and maintain a model serving stack?"

LLMs are one of the greatest inventions in the AI field and a milestone on the path toward artificial general intelligence (AGI). But there is a lot of nuance when you actually apply it in business.

First, there are cost considerations. Businesses hardly ever use a single LLM to solve all of their business use cases. In the real world, businesses prefer to use smaller and cheaper models as much as possible and leverage LLMs for complicated requests. Also, when usage picks up, we generally want to try adopting lower-cost LLM serving. For example, running an open source LLM (such as DeepSeek) on an AWS Elastic Compute Cloud (EC2) instance (that is, a virtual server) can be 30% to 60% cheaper than doing so on a managed serving solution like AWS SageMaker. Also, you can often get a good deal (say, a 70% discount) if you reserve an EC2 instance, which further reduces the serving cost.

A typical chatbot app might use a smaller intent-classification model to triage customer questions, an embedding model to encode those customer questions and search for potential answers, and an LLM to break the customer's request down into actions (backend API calls) and generate human-friendly user interactions.

Another aspect is data privacy and security. If your company is handling sensitive or proprietary data, you might be forced to leverage an in-house solution to avoid data breaches or compliance-related issues.

Lastly, most foundation-model vendors either don't provide the functionality to serve your fine-tuned model or charge a much higher price if you need to serve your own fine-tuned version of the model. For example, as we write this book in early 2025, OpenAI charges 50% more to serve a fine-tuned model. At the same time, smaller models are becoming much more powerful and providing fine-tuning for custom

data and requirements, faster inference, better and more consistent results, and a lower cost to serve.

As technology keeps developing, it trends toward affordable and open approaches, and LLM businesses are no exception. There is nothing wrong with consuming LLMs from vendors, but having your own serving solution (on-cloud or on-premises) with an open source LLM will give your business a critical edge at a lower cost.

Question 3: "Building and maintaining our own serving stack is expensive. What would make it worth the cost?"

Reducing model serving costs is an everlasting effort. As the industry develops and technology and infrastructure change, no one can guarantee that simply outsourcing your serving component will always meet your business's goals. Understanding model serving deeply can help you know all the trade-offs in order to make optimal choices. Let's look at a few examples.

First, imagine you run a Series A startup in the education sector. Your app leverages LLMs to grade students' homework and generate personalized study plans. Working with a fully managed LLM provider like OpenAI, DeepSeek, or Anthropic is a great starting point to accelerate development and validate your business model.

However, as your usage scales, the serving costs start to become unsustainable. To optimize expenses, you consider migrating to a customized model serving solution: for example, using open source LLMs, selecting optimized serving frameworks, and fine-tuning inference performance. This could significantly reduce your costs while maintaining a good user experience.

Let's look at another scenario. This time, you run a customer relationship management (CRM) company that leverages LLMs to provide sales recommendations by processing data from multiple sources, including text-based inputs (emails, Slack messages, customer notes), unstructured audio searches (voice calls, meeting transcriptions) and structured CRM databases (past sales records, customer interactions).

Given the high volume of LLM inference requests and the involvement of sensitive customer data, fully outsourcing your model serving could introduce data-security risks and escalate your operational costs. You decide that owning and optimizing an in-house serving stack is necessary to lower the cost per inference, comply with security regulations (like GDPR and HIPAA), and achieve better performance and latency control.

At the same time, building and operating an in-house serving stack requires significant up-front capital investment in specialized hardware, as well as ongoing investment in engineers with expertise in deploying, maintaining, and optimizing that infrastructure. As a result, the decision to build in-house versus outsource depends not only on security and privacy requirements, but also on the maturity of the

project, its expected scale, and the organization's ability to sustain the operational complexity over time.

Our experience has been that, while outsourced solutions can be a great starting point, businesses must continuously assess whether in-house serving provides a long-term competitive advantage in terms of cost savings, security, and flexibility.

There Is No One-Size-Fits-All Solution for Model Serving

The authors of this book have been building ML systems for a variety of enterprise use cases for over a decade. We've worked on projects where models are fully outsourced (OpenAI), self-managed on-premises solutions (TensorFlow serving, TorchServe, NVIDIA Triton, Ray), fully managed cloud vendor solutions (AWS Bedrock), and deeply customized vendor solutions (AWS SageMaker).

What we've learned over the years is there is no one-size-fits-all or "eternal" architecture. Model algorithms and serving technologies keep evolving, and businesses need to keep pace. Adopting the new technology proactively lets you provide a better user experience and reduce operational cost, helping to keep the business alive and help it thrive.

As a business owner or model serving developer, it's mandatory to have a solid understanding of the fundamentals of model serving so that you can make sense of new innovations in the industry and assess pros and cons to make the right technical choices.

At the end of day, we don't want you to be locked into any specific frameworks or vendors: we want you to be able to understand and adopt new serving technologies quickly in order to give your business an edge among fierce competition.

Why Optimize Model Serving (Especially for LLMs)?

Once a model is successfully deployed, it may appear "done" from a functional standpoint—it produces correct outputs and responds to requests. However, for LLMs, a naively deployed serving system often fails quickly in practice. Latency grows under load, throughput plateaus far below hardware capacity, and serving costs scale linearly—or worse—with usage. In many cases, a serving setup that works in a demo or pilot becomes economically or operationally infeasible once real users arrive.

The next challenge after model hosting is cost optimization. This raises key questions: Can we run the model on more cost-effective hardware? Can we improve *throughput* (the number of model predictions processed per unit of time) and reduce *latency* (the time it takes to process a single model prediction) without upgrading infrastructure?

Can we fully utilize the cloud vendor's LLM service? This is where model serving optimization comes into play.

Model serving optimization refers to the process of improving model serving performance, such as reducing serving latency, increasing throughput, and optimizing resource utilization. In practice, we utilize serving optimization to maximize serving efficiency while keeping costs under control.

Optimization Is Critical for LLMs

LLMs are large and complex enough to demand significant computing power, which can make their operational costs unaffordable for small businesses. For instance, Alphabet chairman John Hennessy told Reuters in 2023 that "running an LLM request can be 10 times more expensive than a traditional keyword search, potentially leading to billions in additional costs."

So, for LLMs, it's almost a must to optimize your model serving: doing so lets businesses reduce costs, enhance user experience with faster response times, and gain a competitive edge over companies with slower or more expensive inference solutions.

In the next section, we'll look at an example to help you get a feeling for why an optimized model and serving system—especially for LLMs—can make a huge difference compared to a "raw" model straight from training.

No worries if some of this feels unfamiliar! We've got you covered—Chapters 5 through 8 will dive deeper into these concepts with hands-on examples to make everything clear, so stay tuned.

Example: Using a Model Serving Framework (vLLM) to Improve LLM Throughput

To get a good grasp of model serving optimization, it helps to first understand the difference between a model *training framework* and a model *serving framework*. A common question people ask is: "Why not just use the same setup for serving as we do for training? After all, both involve running the same model, right?"

Technically, that's partially true—both processes involve loading the model and executing its architecture. But the goals and requirements are completely different. Table 1-1 breaks them down.

Table 1-1. Fundamental differences between model training and serving frameworks

Aspect	Model training	Model serving
Stage in ML lifecycle	Prepares the model	Deploy to production
Objective	Run the model as a process to learn parameters (weights) by minimizing a loss function on the training data.	Run the model to generate prediction (inference) on new input efficiently.
Computation	Extremely compute-intensive, involving iterative model weight updates (including backpropagation and gradient updates)	Focused on efficient forward propagation only (no backpropagation and model updates)
Throughput and latency	Preferable for high throughput (many samples per second). Large data batches are usually processed in parallel to optimize GPU utilization.	Optimized for low-latency response per request; often operates on single or small batches
Resource requirements	Requires powerful GPUs/TPUs and distributed training frameworks (such as Fully Sharded Data Parallel [FSDP] and DeepSpeed). A scale example is the Llama3 405B model that was trained on 16,384 NVIDIA H100 GPUs.	Optimized for low-latency, resource-efficient execution, often on CPUs, edge devices, or inference-specific GPUs (such as NVIDIA TensorRT and ONNX Runtime)

From Table 1-1, you can see why using a model-training framework and system for serving would be inefficient. Instead, we adopt model serving-specific frameworks for serving, such as NVIDIA Triton Inference Server, vLLM, and SGLang.

Now, let's review an LLM inference example with the vLLM serving framework, an optimized LLM serving engine designed by UC Berkeley's Sky Lab for high-throughput, low-latency inference. Using frameworks like vLLM to host LLM models can dramatically improve LLM serving efficiency and throughput compared to baseline approaches such as Hugging Face Transformers or PyTorch's native execution. These baseline frameworks are primarily designed for model training.

Like other model serving frameworks, vLLM exposes lots of knobs for engineers to tune for optimal model performance. Take a look at the following vLLM command for running the OpenAI gpt-oss 20b model on a H100 80 GB GPU, with a few optimal configurations:

```
python -m vllm.entrypoints.openai.api_server \
--model openai/gpt-oss-20b \
--dtype bf16 \
--gpu-memory-utilization 0.9 \
--max-num-seqs 16
--max-num-batched-tokens 16384 \
--tensor-parallel-size 2
```

This vLLM command passes several configurations to vLLM aimed at achieving good throughput and latency on the NVIDIA H100 GPU. For example, it sets PagedAttention (which is enabled by default in vLLM) to KV Cache, sets model precision to `bf16`, sets the max concurrent request (batch size, `max-nums-seqs`) to 16, and shares the model across two H100 GPUs (`tensor-parallel-size`). We'll talk about all of these options in Chapters 6 and 7, so it's OK if they are unfamiliar to you.

The serving performance improvement could be impressive with the right setup. The vLLM team experimented in 2023 by adopting PagedAttention on two settings: Llama-7B on an NVIDIA A10G GPU and Llama-13B on an NVIDIA A100 GPU (40 GB).

They achieved throughput up to 24 times higher than HF Transformers and up to 3.5 times higher than HF Text Generation Inference (TGI). Figure 1-3 illustrates their findings.

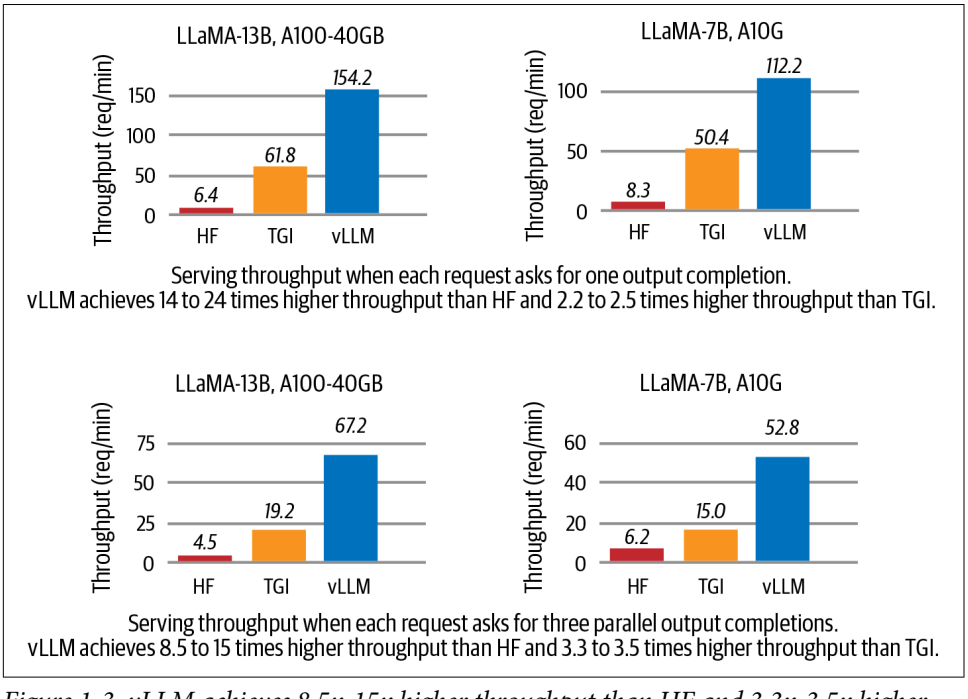

Figure 1-3. vLLM achieves 8.5x-15x higher throughput than HF and 3.3x-3.5x higher throughput than TGI (adapted from Kwon et al., 2023)

I want to emphasize just how big an impact model optimization can have—especially on LLMs. Given the high cost of GPU devices, achieving 6x throughput and 3x lower latency without any additional infrastructure cost is a game-changer.

For production LLM systems, optimization is a must have, and it is not primarily about squeezing out extra performance—it is about meeting business SLAs for latency and throughput at a sustainable cost.

Your Serving Framework's Configuration Impacts Performance Significantly

Every option available in a serving framework represents a potential optimization technique, and the right configurations can make a huge difference in performance.

For example, when we experimented with hosting the DeepSeek R1 model with vLLM, we gained a 15x increase in throughput—jumping from 38 tokens per second (TPS) to 600 TPS—by doing two simple things in vLLM's configuration: enabling FP8 Matrix Learning Accelerator (MLA) kernels and increasing the batch size.

Model Serving Paradigms

In this section, we'll explore the most common model serving paradigms and their real-world applications. We'll introduce these designs progressively—starting with on-device and single-model serving, then expanding to more complex multi-model serving architectures, each building upon the previous concept.

Along the way, we'll share insights from our real-world experience to help you understand the trade-offs of each approach. By the end of this section, you'll have a clear sense of which serving solution best fits your business needs. The practical design principles, along with hands-on examples of model serving systems, will be covered in Chapters 2, 3, and 4.

On-Device (Edge) Serving

On-device serving refers to running the model directly on user-side devices, rather than relying on remote web services. This setup allows applications to process data locally—where it is generated—on devices, such as smartphones, drones, robots, cameras, and VR headsets, enabling real-time computation without the need for constant internet connectivity.

On-device serving has four key benefits:

Cost-effective
: Running models locally on devices reduces dependency on cloud infrastructure, which could significantly reduce recurring operational costs. For example, a smart speaker performing voice recognition locally avoids constant cloud API usage fees.

Efficient
: Leveraging on-device compute speeds up response times and improves power efficiency, especially on hardware optimized for AI workloads. For example, mobile photo-enhancement apps that use on-device inference provide real-time results without latency.

Private

Since data never leaves the device, on-device serving strengthens security and preserves user privacy. For example, for a health-tracking app, analyzing personal biometric data locally allows it to avoid transmitting sensitive information to external servers.

Personalized

Models can be continuously updated and fine-tuned with user-specific data directly on the device, without needing to send data to the cloud. For example, imagine a keyboard app that learns your typing style over time to improve suggestions—without uploading your keystrokes.

Given these benefits, on-device model serving has been widely adopted across various applications.

On-device serving design

Now, let's explore how on-device model serving operates. We'll begin by examining the key components of an on-device AI application, followed by an explanation of the deployment process required to prepare a model for running on different devices. The core concepts are illustrated in Figure 1-4.

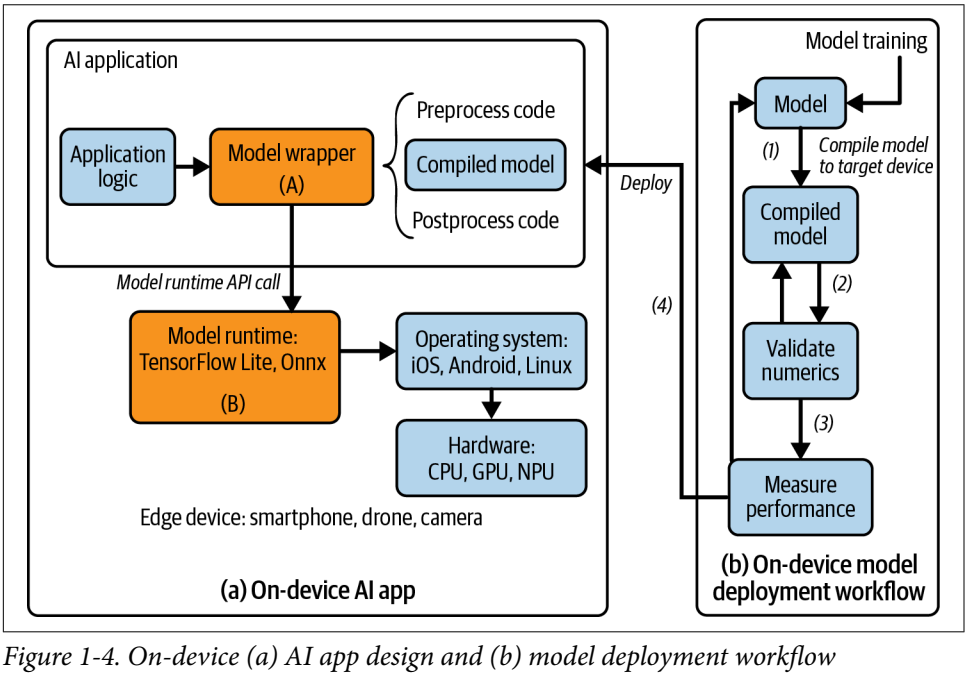

Figure 1-4. On-device (a) AI app design and (b) model deployment workflow

At a high level, the model runtime and model wrapper are the key components of an on-device AI app design, shown in Figure 1-4(a) as components A and B.

A *model runtime* is a specialized software framework designed to execute ML models efficiently on different hardware platforms. It serves as the intermediary layer between the trained model and the device's hardware, optimizing the inference process to maximize speed, efficiency, and resource utilization.

The model runtime plays a crucial role in on-device AI by abstracting the complexities of model execution across different hardware (like smartphones, drones, and robots) and operating systems (iOS, Android, Linux). It also provides hardware-specific optimizations and acceleration techniques to improve inference performance. For example, most model runtimes support *delegates*—a software component that offloads certain parts of model execution (usually compute-intensive operations) from the CPU to specialized hardware like a GPU to optimize performance.

A well-designed model runtime allows developers to focus on building application features rather than worrying about hardware compatibility or model execution details. As of this writing, some of the most popular model runtimes include LiteRT, ONNX Runtime (ORT), and Core ML.

As AI app developers, we use model runtimes to execute models efficiently and encapsulate model-specific execution logic into a reusable module, often called a *model wrapper*; see component A in Figure 1-4(a). This wrapper abstracts low-level details, making it easier to integrate, update, and optimize models within applications.

A *model wrapper* is a component implemented by the application developers to provide the model inference interfaces (functions) for the application logic to execute the model. The model wrapper encapsulates the details of how to interact with the model runtime, such as preprocessing the input data to model input, loading the model into the model runtime, executing the model from the model runtime, and postprocessing the output.

In the workflow shown in Figure 1-4(a), the application logic first asks the model wrapper to execute a model with given data. The model wrapper takes care of data conversion and calls the model runtime to execute the model. The model runtime then handles the execution using the local hardware.

Preparing a model for on-device serving

Figure 1-4(b) describes the high-level workflow of getting a model ready, from the training process to operating on-device.

Once the model has been trained, you first need to convert it from its training format into the format the model runtime requires. For example, if you're using the LiteRT runtime, you can convert (compile) a Resnet PyTorch model into LiteRT's file format (*.tflite*) by using the following command:

```
ai_edge_torch.convert(resnet18.eval(), sample_input)
```

Once the model is compiled, the next step is to validate its numerical accuracy on the target device. This ensures that converting the model and optimizing it for on-device execution haven't degraded its accuracy. The most common validation method is to run the same model inputs on both the training server and the local device, then compare the output differences.

Next, you'll need to measure the model's performance to ensure that the local hardware can execute the model efficiently while meeting the business requirements. Once you've validated that, you can package the optimized model as part of the application installation or update to release to customers.

This deployment process may seem tedious, but tools like Qualcomm AI Hub help automate all the key steps, including on-device model compilation, optimization, and validation, making the workflow much more efficient.

When on-device serving is the right choice

Running models directly on devices offers many benefits like low latency, offline capabilities, and better privacy. Here are some use cases that favor on-device hosting:

Privacy-first workloads
: On-device serving is ideal when raw data must never leave the user's device—such as biometric authentication, personal health data, or sensitive user interactions. By running inference locally, systems can meet strict privacy and compliance requirements without relying on remote services.

Ultra-low-latency applications
: For applications where milliseconds matter—gesture recognition, robotics control loops, AR/VR, or real-time audio processing—on-device inference avoids network latency entirely, enabling deterministic and responsive behavior.

Disconnected or intermittent environments
: In scenarios with unreliable or unavailable connectivity, such as industrial equipment, remote sensors, vehicles, or drones, on-device serving ensures the system continues functioning independently of the network.

Internet of Things (IoT), smart cities, and robotics
: Edge devices in smart infrastructure and robotics are often designed around specific workloads. On-device models are optimized for power efficiency and task specificity, making them well suited for continuous inference under constrained resources.

Local devices operate under strict constraints in compute capacity, power, storage, and update mechanisms. These constraints don't make on-device serving inferior—

but they strongly influence which models are practical to deploy and when hybrid or remote serving becomes necessary. Here are some trade-offs and constraints:

Computation and storage constraints
: Smartphones, IoT hardware, and embedded systems have limited CPU, GPU, and memory resources, making it difficult to run large, complex models. Although there are models trained to run on edge devices, such as Gemma 3 270M (Google) for on-device deployment, this isn't feasible for all models due to memory, storage, and processing-power constraints.

Power consumption
: Running AI models, especially deep neural networks, can be power-intensive, draining battery life quickly. Usually on-device models are designed to be power efficient for the task they are designed for, but there are always limits. For example, we don't run real-time video enhancement (from low to high resolution) locally since it consumes excessive amounts of power.

Limited update and maintenance capabilities
: On-device models are hard to update frequently. When improvements are made to the model, each device must receive and install an update. For example, a banking app using an on-device fraud-detection model would need to release frequent app updates to improve its model's accuracy.

Inconsistent hardware support
: Different devices have different hardware capabilities: some support NPUs, others only CPUs. This can make it hard to deploy optimized models across all devices. For example, a model optimized for Apple's Neural Engine on an iPhone might not run efficiently on an Android device with a Qualcomm Snapdragon processor.

When on-device constraints begin to outweigh their benefits—due to model size, traffic volume, or operational complexity—serving naturally shifts to on-premises or cloud environments. These environments trade local execution for centralized compute, elastic scaling, and simplified operations. The most common starting point for server-side serving is the single-model service pattern.

Single-Model Service

In this section, we'll introduce you to the classic single-model service design pattern, which is also the most widely used cloud-based model serving approach.

In this design (diagrammed in Figure 1-5), each model and each version of the model is deployed as a dedicated web service, exposing a prediction API over HTTP or gRPC for server-side execution. This setup ensures scalability, isolation, and flexibility in model deployment.

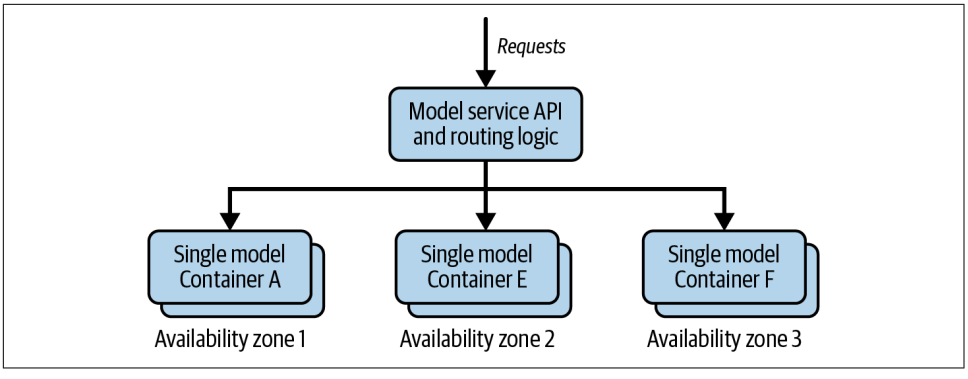

Figure 1-5. Single-model service architecture

From Figure 1-5, you can see that the single-model service architecture follows a standard microservices design. It has an API component in front to receive web prediction requests and route them to one of its backend workers or containers. The backend workers perform the actual model execution and return the result.

Containerization Is the Foundation of Modern Model Serving

Containerization means creating an isolated environment (or *container*) that packages an application along with all its dependencies, libraries, and configurations. Using a container ensures that the application runs consistently across different computing environments, whether on a developer's laptop, a test server, or in production.

In modern model serving, most server-side model execution happens inside a container. It's a common practice to encapsulate the model management, resource management, and model execution logic into a Docker container (or Kubernetes Pod) and expose the model's prediction/serving function via a gRPC or HTTP interface.

Docker is the most popular containerization technology as we write this book, so we use Docker containers throughout the book, including code examples.

Now, let's look at Figure 1-6 to see what's inside a single-model serving container.

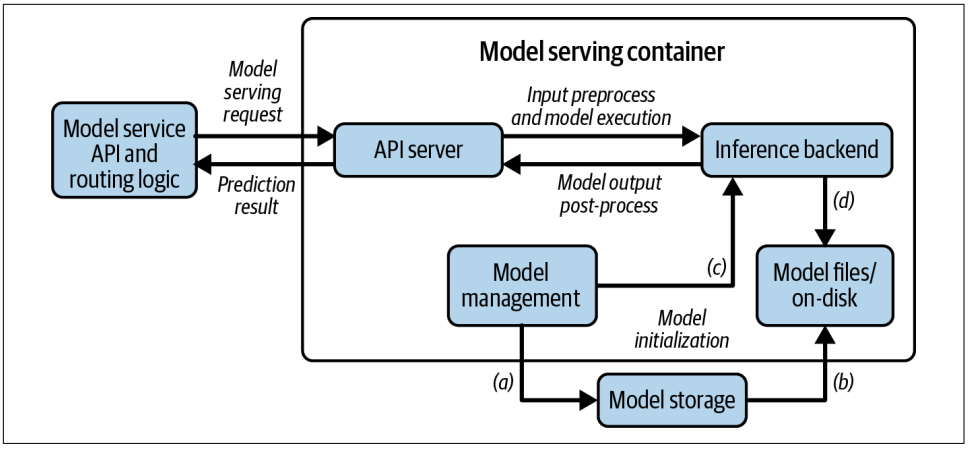

Figure 1-6. Single-model serving container design

You can see that there are three main components in the model serving container (Figure 1-6): API-Server, Model Management, and Inference Backend. *Inference Backend* is the model execution component. It usually leverages a model serving framework (like TensorFlow Serving or TorchServe) or libraries (like vLLM or TensorRT-LLM) to run model serving. *Model Management* is the component responsible for preparing the model for serving. This includes downloading the model, extracting it to local storage, and loading it into the inference backend; see initialization steps (a) through (d) in Figure 1-6. *API-Server* exposes the model inference functionality over HTTP or gRPC, so external applications can send prediction requests.

For model deployment, we expect the model-training pipeline or the Ops team to upload the trained model to cloud storage. From there, the Model Management component detects the new model and automatically refreshes the model in each serving container, ensuring that the latest version is deployed seamlessly.

Routing choices in single-model serving

One of the key responsibilities of the Model Service API component (in Figure 1-6) is to ensure that all the model serving containers are saturated with the prediction workloads *equally*. This sounds like a problem that can be solved easily using a "round-robin" load-balancing strategy. For example, if you have three containers (c1, c2, and c3), the first prediction request would go to c1, the second to c2, the third to c3, and then it starts over again.

However, round-robin routing has a key limitation—it doesn't account for variations in request processing times. In many cases, prediction requests have different computational costs based on input size and payload complexity. For example, an LLM inference request with a large text payload (for example, 5,000 tokens) takes longer to process than a 100-token input. An image model processing high-resolution images consumes more GPU or CPU resources than one processing smaller images.

Round-robin load balancing creates inefficiencies because of this variation: for instance, you might have a situation where container c1 is still processing a large request and container c2 has already finished its work, but the next request is still routed to c1, creating a bottleneck.

To address this issue, we need more sophisticated routing strategies that take into account processing time, resource availability, and request complexity. Here are some frequently used routing choices:

Weighted round robin
: Servers (bigger) with higher weights receive more requests

Least connections
: Directs requests to the server with the fewest active connections

Least response time
: Routes requests to the server with the lowest response time

Dynamic load balancing
: Uses metrics like CPU, GPU, memory usage, or queue length to distribute requests

Horizontal and vertical scaling

As prediction-request traffic increases, a single serving container may struggle to process requests quickly. So we need to deploy more instances of the serving containers (we'll refer to them as just *instances*) across multiple machines to handle the growing number of concurrent users with minimal delays. This is called *horizontal scaling*, or *scaling out*.

In a production serving environment, we enable *autoscaling*, so that the infrastructure management system automatically provisions more instances based on server resource utilization. The system monitors key metrics such as CPU and memory usage, inference latency, and number of active requests.

When monitored metrics exceed their predefined thresholds, the infrastructure management system spins up additional instances dynamically. For example, if you use Kubernetes to manage your compute cluster, its Horizontal Pod Autoscaling (HPA) can automatically adjust the number of serving pods based on real-time demand.

For large deep-learning models, such as GPT-4 and Llama-2-70B, a single machine may not have enough GPU memory to load the model. In these cases, we use *vertical scaling* (*scaling up*). This means deploying the model on more powerful GPUs (such as NVIDIA H100s or A100s) and using distributed serving across multiple GPUs or multiple machines to split the workload.

Distributed model serving can be complex, but modern frameworks like vLLM simplify the process by abstracting away the low-level details. You can easily specify the number of GPUs to use when launching a model. For example, to serve a Llama model on 4 GPUs using vLLM, you can run this command:

```
python -m vllm.entrypoints.api_server \
   --model meta-llama/Llama-2-13b-hf \
   --tensor-parallel-size 4 \
```

Prioritizing Intra-Node Serving (Multiple GPUs on One Machine) over Inter-Node Serving (Multiple Machines)

For better serving performance, easier management, and lower latency, it's generally preferable to run large models on a single machine with multiple GPUs (*intra-node serving*) rather than to distribute them across multiple machines (*inter-node serving*).

Inter-node serving introduces network overhead, increasing latency and complexity in synchronization. Whenever possible, we recommend you optimize model execution to fit within a single machine, to maximize efficiency.

We'll dive deeper into these model optimization techniques in Chapters 6 and 7.

Why single-model service is the default starting point

The single-model service approach is our favorite because of its simplicity and versatility. It works seamlessly with any type of model, small or large, from LLMs to deep learning models, making it a reliable and scalable choice for model serving.

For production and operations, we use single-model service whenever possible for the following reasons:

- It offers the best performance, with no resource contention and lower latency.
- Scaling is easy, since each model can scale independently.
- Deployment and debugging are simpler thanks to isolated logs, metrics, and updates.
- It's more reliable since if one model crashes, it doesn't affect others.
- Optimized hardware per model allows for better cost control.

Although the single-model serving approach is great, it falls short in some areas, particularly in resource efficiency and cost.

Imagine you're running an agent platform where customers can create their own agents and deploy custom models. If you have 100 customers and each deploys 10 models, that means your platform would be hosting 1,000 separate model services. This setup isn't scalable—the overhead of maintaining, patching, deploying, and monitoring so many services would be overwhelming. Also, many uploaded models might never be used, wasting compute resources and incurring unnecessary infrastructure costs.

To reduce operational complexity and maximize resource utilization, you need a more efficient solution—one that allows multiple models to share compute resources dynamically, instead of running a separate service for each.

This brings us to multi-model service, a design approach that enables cost-effective, scalable model serving.

Multi-Model Service

Multi-model service is designed to co-host multiple models in one serving container, sharing GPU/CPU and memory across multiple models, and to load and unload models dynamically based on the incoming traffic. This can help you save significantly on costs and achieve the best price performance.

Multi-model service is an ideal solution when you need to serve a large number of models efficiently without loading them all into memory at once. Instead of dedicating a separate container to each model, this approach has models *share* compute resources within a single serving instance.

Going back to our agent-platform use case, discussed in "Why single-model service is the default starting point" on page 23, instead of loading all 1,000 models into GPU and memory simultaneously, with multi-model service, you could adopt an on-demand approach. Load the model only when a customer requests a prediction, and unload it when it's inactive or when memory needs to be freed up for a new model. This significantly reduces infrastructure costs and improves resource utilization, making the model serving far more scalable.

You might wonder: how is it possible to host multiple models in the same serving container when they may have been trained using different frameworks and require different execution parameters?

The answer lies in the design of the model serving container, which manages model loading, execution, and unloading dynamically. Let's explore its architecture, shown in Figure 1-7.

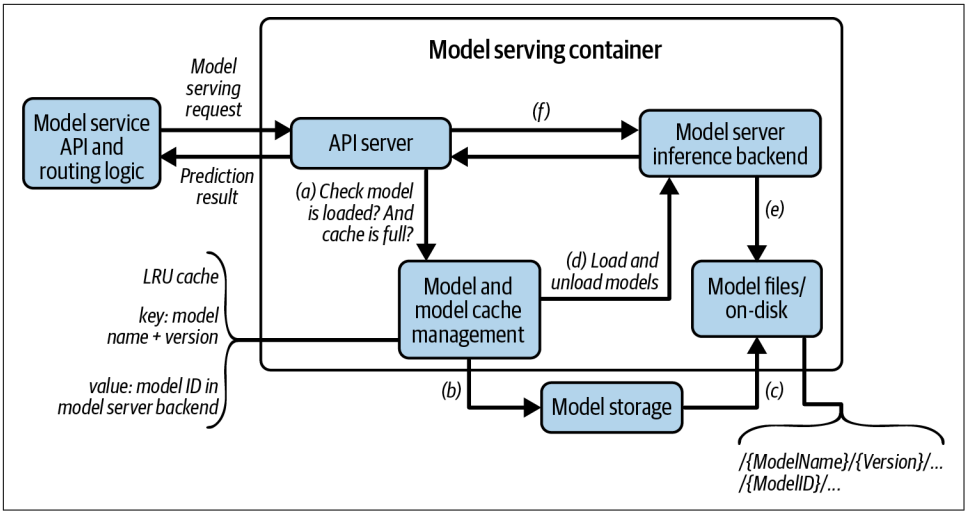

Figure 1-7. Multi-model serving container design

If you compare Figures 1-6 and 1-7—the single-model and multi-model serving containers—note that the multi-model serving container has *two* different components: a model server inference backend and model cache management.

The *model server inference backend* is designed to handle multiple types of models in a black-box manner. Regardless of the model's algorithm and version, the model server inference backend can serve these models with a unified web prediction API. The trick here is that the model server contains multiple types of inference backends internally, and each backend handles one type of model: for example, TensorFlow, ONNX, or PyTorch.

Fortunately, you don't have to build a model server from scratch—there are robust and generic open source solutions available, such as NVIDIA Triton Inference Server. In practice, we deploy systems like Triton inside model serving containers (or Kubernetes Pods), using Triton as the model server inference backend to efficiently host and manage models with different requirements, such as GPU, memory, and frameworks.

The *model cache management* component has two responsibilities. First, it loads a given model from model storage into the model server inference backend. Second, it prevents the container resources from model overload by maintaining a least recently used (LRU) cache to track all the modeled models, so it can unload the least-used models when the container's resource utilization is high.

Triton Inference Server: A Unified, Multi-Model Serving Platform

Triton Inference Server is one of the most popular open source model serving solutions. It's also one of our go-to tools, especially for multi-model use cases.

Triton can manage and serve any number or mix of models, constrained only by disk space and memory resources. It provides a unified API on top of various model formats, including TensorRT, TensorFlow GraphDef, TensorFlow SavedModel, ONNX, PyTorch, Caffe2 NetDef, and more. See more about Triton from the Triton Architecture Blog.

Now, let's look at an end-to-end on-demand model serving workflow. When a prediction request on model A hits the container, model management will do three things for the request:

- If model A is loaded already, it forwards the request to the model server backend to execute the model and return the result.
- If model A is not loaded, the management component will download the model from model storage, load it into the model server, andModel Server Inference Backend forward the request.
- If memory and disk consumption is above a specified threshold (for example, 80% memory usage), the management component will find the least used model in its cache, unload it from the server backend, and remove the model from cache and disk.

Routing and autoscaling in multi-model service

Using shared resources to host multiple models is efficient and cost-effective, but it also introduces challenges in routing: how do we send different model prediction requests to the right serving container and scale them effectively? Let's dive into these challenges and explore a potential solution.

You can see two problems in Figure 1-8. The first problem is request routing. Since models can be hosted in any of the serving containers, we want to route the prediction request to the container that has already loaded the model. Otherwise, we add extra latency to the request, either from a *cold start* (holding the request in waiting until the model is loaded into memory) or *model swapping* (removing old models out of memory for the new requested model).

The second problem has to do with per-model scaling. Not all models will get the same amount of traffic; some models will be used more frequently than others. We want to have more model instances in our multi-model containers for these "hot" models.

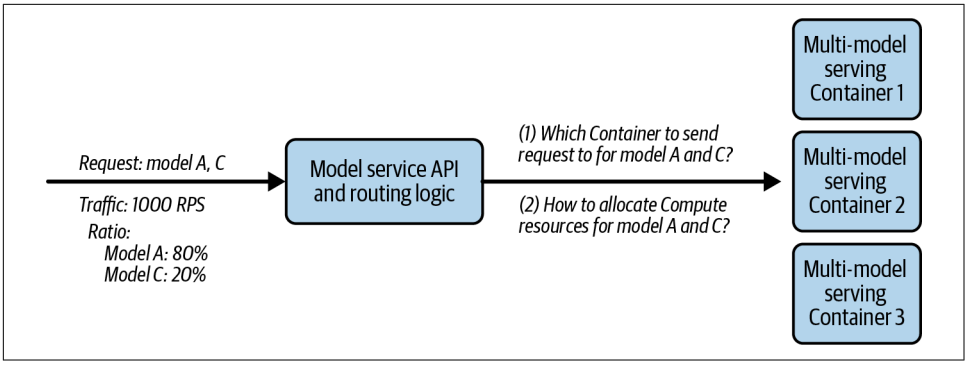

Figure 1-8. Serving and scaling challenges in multi-model serving

To solve the routing and model-scaling issues, we need to enhance the routing component at the frontend API layer (see Figure 1-9).

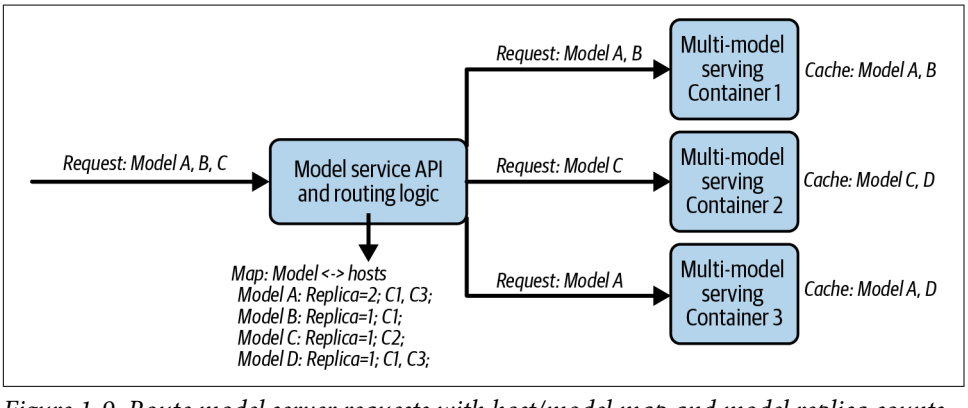

Figure 1-9. Route model server requests with host/model map and model replica counts

In Figure 1-9, we add two features to address the aforementioned routing and scaling problems. First, we add an attribute `replica` to the model's metadata. The value of a model replica defines how many instances of this model will be hosted in the backend containers. Second, we introduce a route map in the routing layer and use it to track to which containers a given model prediction request should be routed.

For example, based on the `Map: Model <-> hosts` section in Figure 1-9, for a given Model A with a replica count of 2, the routing logic will choose two containers to host model A: in this case, containers C1 and C3. Then the routing component will route all future model A requests to C1 and C3 evenly. The router can also adjust model A's replica count up and down in real time to match model A's traffic. When adding new replicas to model A, it's also the router's job to choose the right serving container to host the newly added model instance (it's like a bin packing problem).

When multi-model service becomes necessary

As we've explained, multi-model service is the most cost-efficient serving method because it lets you use a shared resource pool to host more models. It's extremely powerful not to have to host all the models at the same time.

However, multi-model service tends to struggle in the following situations:

- When the models are too large to fit in one GPU, either because there is no room to share or because lots of model unloading is required to make memory space available. This causes a lot of model-loading overhead and cold-start latency for future models.
- When individual models expect high traffic that requires low latency and the model always being loaded. Single-model service is better in these cases.
- When the models have different security policies.
- When operational complexity is high: for instance, deployment, debugging, and monitoring are complicated due to the number of moving pieces in the system, such as model cache management, per-model routing and scaling, model compatibility, and dependency conflicts.

You might have noticed that the multi-model and single-model serving approaches are complementary. In practice, platforms combine multiple serving approaches to address different serving use cases under the same roof. The next section discusses model serving platforms.

Model Serving Platforms

As a business grows, model serving demand surges and the requirements become increasingly complex. Simply putting single-model and multi-model services together is no longer good enough. These serving systems face two major challenges.

First, many tasks now require multiple models to work together. For instance, a voice assistant like Apple Siri must integrate speech recognition, natural language processing (NLP), recommendation models, and text-to-speech synthesis to respond to a single voice command. Managing these multi-model interactions efficiently is crucial for seamless performance.

Second, when a growing number of AI applications are running on one platform, optimizing compute resources becomes a critical challenge. You need an efficient strategy to allocate GPUs, CPUs, and memory across models while ensuring scalability and cost-effectiveness, and minimizing resource contention.

The model serving platform design in Figure 1-10 gives you a sense of how a model serving platform addresses the aforementioned challenges. First, it uses resource groups to segregate the prediction workload for different applications and business

scenarios. For example, there are two single-model services and one multi-model service setup in the App1 resource group. Each resource group is allocated its own CPU, GPU, and memory quotas, which is based on business requirements to optimize performance and cost.

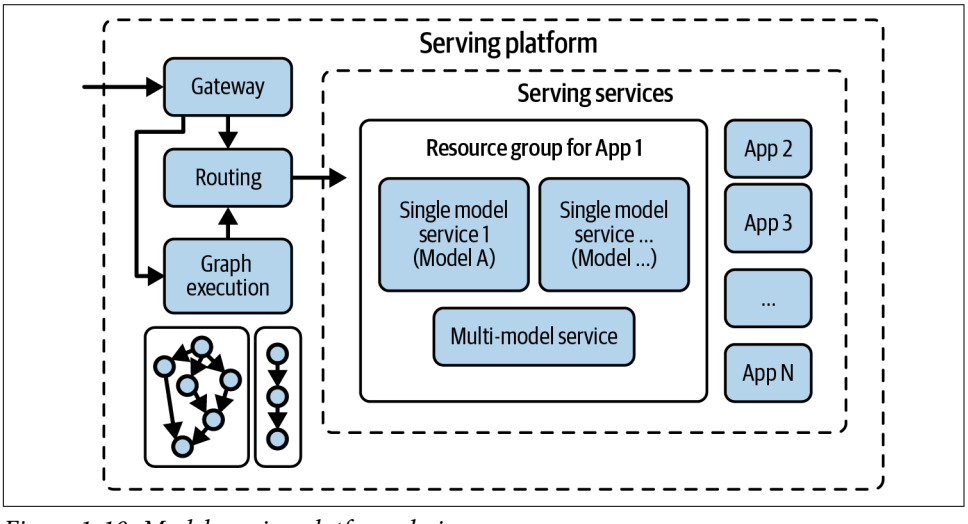

Figure 1-10. Model serving platform design

Second, the design has a graph execution component (for example, Airflow or Ray) to support multistep inference workflows. Each app team defines its own inference workflow and deploys it to the graph execution engine. During the prediction time, the graph engine first gets the request and runs the predefined inference workflow. For each inference step in the workflow, the graph engine calls the routing component to find the right serving group and service to run the model inference.

Figure 1-10 provides a high-level overview of a model serving platform. In practice, this design could be a lot more complicated. For example, a serving platform must have security and access-control components, metric and monitoring components, deployment and DevOps integration, and more. Since these components are not core to model serving, we've omitted them from this diagram to keep the focus on the main serving concepts.

Summary

In this chapter, we explored the structure of a model from an engineering perspective, viewing it as executable components rather than just static data files. A model consists of three key elements: model data (weights, biases, and configurations), model architecture (the structure of layers and operations of the model), and model execution code (which loads and runs the model for inference).

Next, we introduced model serving, which involves deploying models in production environments to process real-time data and generate predictions. We highlighted three deployment scenarios—on-device, on-premises, and cloud-based model serving—and outlined the key considerations for model serving, such as scalability, latency, monitoring, security, and cost optimization.

Then we explored why understanding and optimizing model serving is essential, even when leveraging cloud vendor solutions or large language models (LLMs). While cloud-based model serving is convenient, businesses must still navigate complex integrations, cost trade-offs, and security concerns. Similarly, LLMs are powerful but often require multi-model pipelines, fine-tuning capabilities, and cost-efficient deployment strategies to be practical in real-world applications.

We also introduced model serving optimization, particularly for LLMs, where efficiency gains can lead to significant cost reductions and performance improvements. By optimizing techniques such as KV Cache with Paged Attention (vLLM) and Radix Attention (SGLang), businesses can achieve six times higher throughput and a third of the latency without incurring additional infrastructure costs. Since LLM inference is computationally expensive, serving optimization is not just an improvement—it's a necessity for cost-effective AI deployment.

Finally, we explored the most common model serving paradigms, from on-device (edge) serving to single-model service, multi-model, and full-model serving platforms. Each paradigm has its strengths and trade-offs, depending on factors such as latency, cost, scalability, and complexity.

We hope this chapter has convinced you that model serving is critical—and that it's an engineering-driven discipline, focusing on efficiently deploying and integrating models into real-world applications, rather than on the intricacies of training models or designing algorithms.

In the next two chapters, we'll dive into hands-on examples to see how model serving is built and operates in practice. We'll start with LLM model serving in Chapter 2, then expand to a broader model serving system design and implementation in Chapter 3.

CHAPTER 2
Large Language Model Serving

In the previous chapter, we introduced the concept of models, model serving, and common serving paradigms. In this chapter, we shift our focus to the specific challenges and techniques involved in serving LLMs.

One of the most common barriers for those entering the world of model serving is the sheer complexity of modern serving systems. With rapidly evolving model architectures, training algorithms, and tooling—combined with layers of production infrastructure like monitoring, scaling, security, CI/CD pipelines, and service dependencies—the experience can quickly become overwhelming. The result is that many engineers and researchers lose focus, getting bogged down in system details before they can grasp the core principles.

To address this, our approach is to start from the fundamentals. We begin with the bare minimum code required to serve an LLM and build up from there. This helps establish a strong mental model of how token generation works and why serving LLMs poses unique challenges. From this foundation, we'll gradually expand into more advanced topics like system architecture, performance optimization methods, and infrastructure choices in the remaining chapters. In this chapter, we start with:

- The basic architecture of LLMs, including the token generation process and attention mechanisms
- What happens under the hood during inference through hands-on code examples
- The core concepts behind LLM serving, such as prefill, decode, and key-value (KV) cache reuse
- Why understanding these fundamentals is critical to diagnosing bottlenecks and contributing to performance improvements

We'll then transition into using a modern serving framework—vLLM—to demonstrate how to improve model serving efficiency. Next, using vLLM, we will introduce key serving methods such as streaming and batching, which are essential to achieving production-grade performance.

This is a foundational chapter in the book. Our goal is to equip you with the practical understanding and intuition needed to reason about LLM serving systems. You'll gain clarity on their behavior, limitations, and optimization opportunities—laying the groundwork for later chapters, where we dive deeper into system design, scalability, and advanced serving optimization strategies.

Throughout the chapter, we provide hands-on examples to help reinforce each concept. And don't worry—deep mathematical knowledge is not required. Since we are focused on serving rather than training, we've abstracted mathematical concepts into intuitive explanations, allowing you to focus on the engineering perspective.

In the next chapter, we'll take what we've learned here and demonstrate how to wrap it into a real-world web service, covering key design decisions and principles.

Inside the Mind of a Transformer

In this section, we'll introduce the essential concepts behind the Transformer model—specifically from a model serving perspective. Rather than diving into training algorithms or academic theory, we take a top-down, conceptual approach: beginning with the history of LLMs, then walking through the LLM generation process, model architecture, Transformer blocks, and finally the attention mechanism.

LLM Evolution

The evolution of LLMs is not just a historical curiosity—it offers critical insights into the model design choices, architectural patterns, and execution behaviors, which are fundamental to inference and optimization workflows.

Language models have evolved from basic rule-based systems to sophisticated neural networks capable of generating coherent and contextually relevant text. This progress has been driven by advancements in model architectures, the availability of large-scale text datasets, and increasing computational power. Figure 2-1 provides an overview of language model development.

The late 2000s marked a turning point in natural language processing (NLP) with the advent of deep learning (for the sake of keeping this brief, we'll skip over early rule-based methods and n-gram models here). A major breakthrough came in 2013 with Word2Vec, by Mikolov et al. at Google, which introduced dense vector representations of words in a continuous space. These embeddings captured semantic rela-

tionships between words and dramatically advanced the field's ability to understand language.

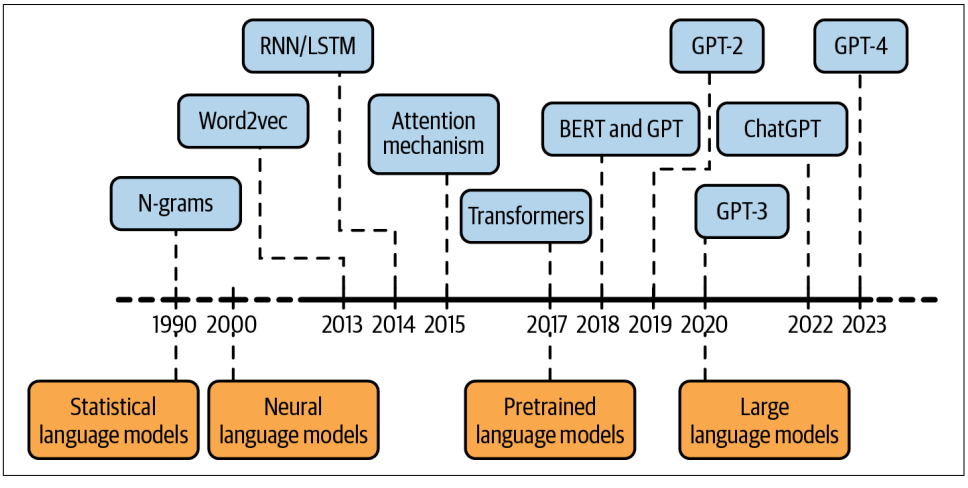

Figure 2-1. The history and development of language models (source)

To effectively model the sequential nature of language, *recurrent neural networks* (RNNs) became popular in 2013. RNNs could capture temporal dependencies and contextual information, making them well-suited for tasks such as sentiment analysis and text generation. To overcome RNNs' struggles with long-term dependencies, more sophisticated architectures like *long short-term memory* (LSTM) networks and *gated recurrent units* (GRUs) were introduced in 2014. These incorporated gating mechanisms and memory cells to better retain and manage information over longer sequences.

Despite these improvements, RNN-based models still had critical limitations. For instance, they processed input sequentially—one time step at a time—making them inherently difficult to parallelize. This constraint hindered their scalability and efficiency on modern hardware like GPUs, and they continued to struggle with capturing long-range dependencies in text.

The introduction of the Transformer architecture, introduced by Google in the 2017 paper "Attention Is All You Need", revolutionized sequence modeling. By replacing RNNs' recurrent layers with self-attention mechanisms and positional encoding, Transformers enable parallel processing while efficiently capturing long-range dependencies.

The introduction of the Transformer architecture sparked the emergence of two influential model families: Bidirectional Encoder Representations from Transformers (BERT) and the Generative Pre-trained Transformer (GPT). A pivotal innovation was the shift from training models from scratch for each task to pre-training on large-scale, unlabeled text using unsupervised learning objectives. This pre-training phase enables models to acquire a broad, generalizable understanding of language, which can then be fine-tuned for specific downstream tasks.

BERT, built on a *bidirectional* encoder, reads text in both directions simultaneously. This makes it particularly effective for understanding context and performing tasks such as text classification or generating contextual embeddings. In contrast, GPT models are based on a *unidirectional* decoder, generating text by predicting the next token given the preceding context—an approach well-suited for generative tasks.

Over time, the GPT architecture has demonstrated exceptional versatility across a wide range of applications, including text generation, summarization, translation, and question answering. Researchers have discovered that scaling up both model size and training data significantly enhanced performance. This scaling unlocked capabilities like few-shot and even zero-shot learning, where models can perform new tasks with little or no additional training.

This trend is evident in the rapid evolution of model scale: GPT-1 had 117 million parameters, while GPT-3 grew to 175 billion within two years. More recently, DeepSeek R1 reached 671 billion parameters. As a result, the term *large language model* emerged to describe these powerful pre-trained models characterized by tens or even hundreds of billions of parameters.

With the brief historical overview out of the way, we now turn to a more technical discussion of Transformer architecture. If you are interested in a deeper exploration of LLM history and development, the paper "A Survey of Large Language Models" offers a comprehensive and insightful read.

In this chapter, *LLM* and *Transformer* specifically refer to *decoder-only Transformer* architectures—models that utilize only the decoder stack from the original Transformer design introduced and popularized by OpenAI's 2018 paper, "Improving Language Understanding by Generative Pre Training".

The Definition of LLMs Will Keep Evolving

As of the time of writing, the term LLM typically refers to *generative, decoder-only Transformer* models. However, it's important to understand that this definition is not static—it continues to evolve alongside advancements in AI research and deployment.

At its foundation, LLMs are built on the concept of large-scale language modeling. But what qualifies as "large" has changed dramatically over time, as have the architectures, capabilities, and training methodologies involved. For instance, in 2018, models like BERT—with a few hundred million parameters—were considered large. By 2023–2024, models such as GPT-4 and Claude 3 are estimated to scale to trillions of parameters and include multi-modal capabilities.

Alongside this growth, the expectations for LLMs have expanded significantly. They are no longer judged solely by their ability to generate coherent text, but also by their capacity to follow complex instructions, interact with tools, and exhibit human-aligned behavior.

Therefore, readers should remain mindful that both the definitions and the expectations surrounding LLMs will continue to evolve as the field progresses.

The Autoregressive Nature of Transformers

In this section, we'll take a high-level, conceptual look at how a Transformer performs text generation given an input prompt.[1] A defining characteristic of LLMs is their *autoregressive* nature: they generate text one token at a time, with each new token predicted based on all previously generated tokens. This step-by-step process allows the model to maintain coherence and context, ensuring that each word aligns meaningfully with what has already been generated. Autoregressive generation also mirrors how language is naturally constructed—progressively, with each word depending on prior context. See Figure 2-2 for an illustration of this process.

1 For simplicity, we use the terms *input text*, *input sequence*, and *prompt* interchangeably throughout the book to mean input to an LLM. We also use the term *token* interchangeably with *word*. However, in practice, a token is defined by the tokenizer algorithm, which determines how text is broken down into smaller units. Depending on the tokenizer and language, a token may represent a whole word, a subword, or even punctuation. On average, one token corresponds to approximately 0.75 words in English. You can explore the OpenAI tokenizer viewer to understand how a piece of text might be tokenized by a language model and the total count of tokens in that piece of text.

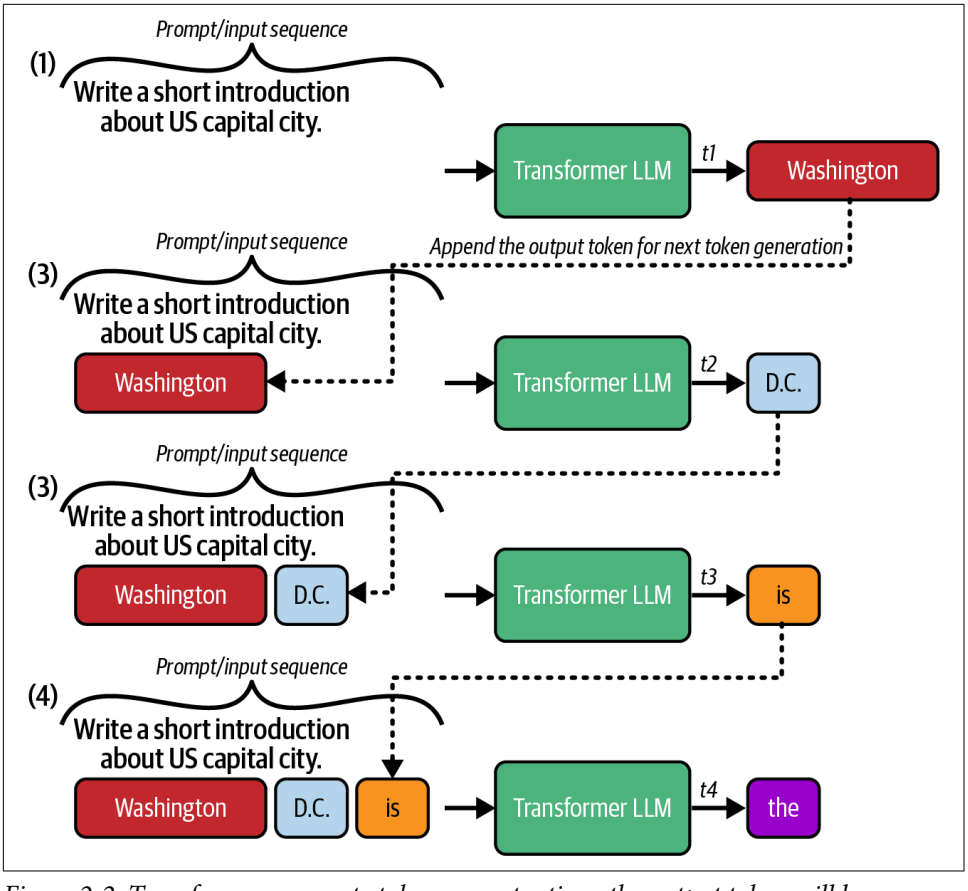

Figure 2-2. Transformers generate tokens one at a time: the output token will be appended to the input sequence as a prompt for the next token generation

In Figure 2-2, you can see how a Transformer-based LLM generates text in an autoregressive manner—predicting one token at a time, each conditioned on the prompt and all previously generated tokens:

- In step 1, the model receives the initial prompt, "Write a short introduction about the US capital city," and generates the first output token: `Washington`.
- In step 2, this generated token is appended to the original input, and the updated sequence is fed back into the model to produce the next token: `D.C.`

- In step 3, the prompt now includes both `Washington` and `D.C.`, allowing the model to generate the next token: `is`.
- In step 4, the sequence continues with the addition of `is`, leading the model to generate `the`.

This iterative process continues until the model encounters a stopping condition—such as reaching a maximum length or generating a special stop token—gradually building the output sequence token by token.

Decoder-Only Transformer Architecture

Now that you've seen how a Transformer model generates text, let's take a closer look inside the Transformer LLM box shown in Figure 2-2.

In this section, we'll explore the underlying architecture of decoder-only Transformers, which power most modern LLMs. In addition to reviewing the high-level components and flow, we'll also examine the architecture of a real-world model—Qwen 2.5—to help build your practical intuition for understanding and analyzing LLM structures in practice.

Why Demonstrate Decoder-Only Transformer Architecture?

There are many architectural variants built on the Transformer framework. While these models all share the same foundational principle—stacked Transformer blocks with self-attention—they differ in how their components are structured and applied.

In this book, we focus on the decoder-only Transformer architecture because it is the most widely adopted design for models that perform generative tasks, such as GPT, Llama, and Qwen. If you are interested in exploring other Transformer architectures, such as encoder-decoder models used in tasks like translation and summarization, you can start with the blog "Transformer-Based Encoder-Decoder Models".

Model architecture

In Figure 2-3, you see a high-level view of a typical decoder-only Transformer architecture. It shows how a prompt like "Write a short introduction about the US capital city" is processed by the model to generate a token such as `Washington`. The model can be conceptually broken down into three key components: tokenizer, Transformer (decoder) blocks, and language modeling (LM) head.

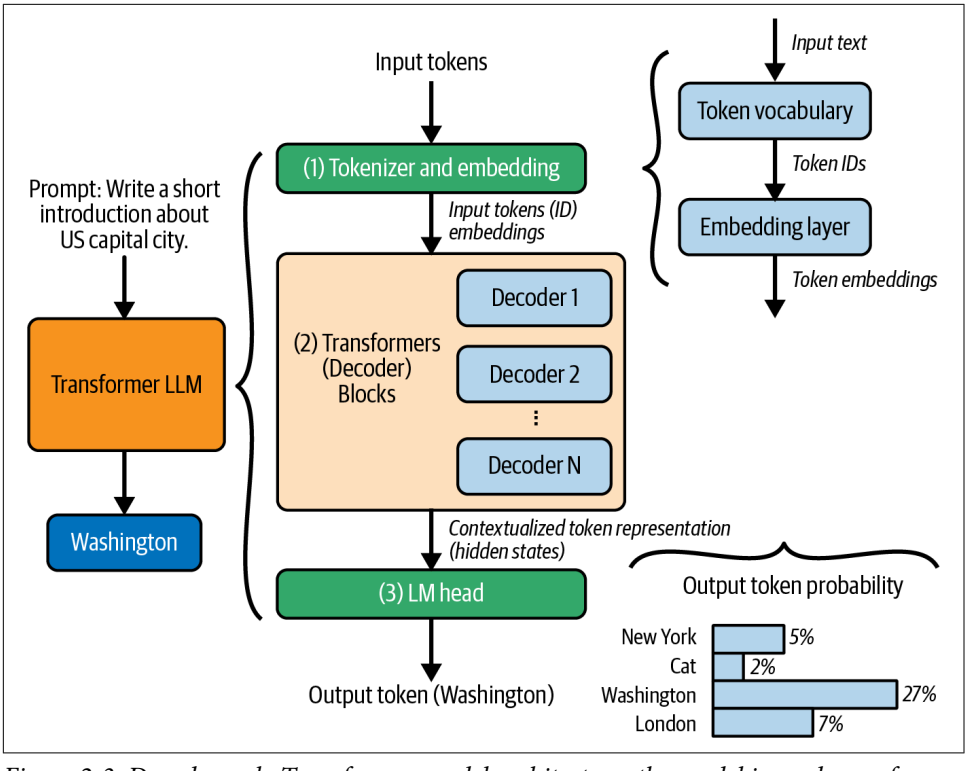

Figure 2-3. Decoder-only Transformer model architecture: the model is made up of a tokenizer and embedding, a list of transformer blocks, and an LM head

Let's examine these components one by one:

Step 1: Tokenizer and embedding

The first step in the LLM token generation is handled by the tokenizer, which takes raw input text (prompt) and does the following:

- Breaks it into discrete tokens based on a fixed vocabulary.
- Converts those tokens into token IDs, which are numerical representations. These IDs map to the actual text in the vocabulary used in step 1.
- Maps those token IDs into vector embeddings using an embedding layer, preparing the input for the Transformer blocks.

This tokenization and embedding step ensures the model can interpret and process human-readable input in a form that the following Transformer blocks will understand.

To better understand the tokenization process, you can explore it using OpenAI's tokenizer tool. For example, given the input text "Write a short introduction about the US capital city," the tool breaks it into 11 tokens: `["\"", "Write", "a", "short", "introduction", "about", "US", "capital", "city", "\""]`. It then maps each token to a corresponding token ID: `[1100, 10930, 261, 4022, 22575, 1078, 290, 2952, 9029, 5030, 693]`. Ultimately, each token ID is mapped to an *embedding vector* (a numerical array). For example, token ID `1100` might correspond to a vector like `[-0.12, 0.55, 0.98, ...]`.

Step 2: Transformer (decoder) blocks

The heart of the model lies in a stack of decoder blocks. These blocks are where most of the computation (and learning) happens during model serving and training.

The decoder blocks are stacked (for example, 12, 24, or more layers), allowing the model to build up a rich contextual understanding of the prompt and previous outputs—token by token—before predicting the next one.

The outputs of these decoder blockers are *hidden states*—contextualized representations of the input tokens. More specifically, they are tensors of shape `[N, d]`, where *N* equals the number of tokens in the input sequence and *d* equals the hidden state size (for example, 768, 2048, 4096, depending on the model).

These hidden states (tensors) are then passed to the next component—the LM head. In most cases, only the final hidden state, corresponding to the last token in the sequence, is used to predict the next token in the output.

Step 3: Language modeling (LM) head

When the LM head generates the output token, it primarily does two things:

- First, it maps the hidden states from transformer blocks to *vocabulary logits*—a probability distribution for the tokens in the token vocabulary.
- Second, it selects the output token based on the vocabulary logits (probabilities). Usually, the token with the highest probability is selected as the next output. For example, in Figure 2-3, the LM head selects `Washington`, followed by alternatives like `London`, `New York`, and `Cat`.

With this foundational understanding in place, let's now examine the architecture of a real-world model—Qwen 2.5—with the following code as a concrete reference:

```
model_name = "Qwen/Qwen2.5-0.5B"
# load Qwen2.5
model = AutoModelForCausalLM.from_pretrained(
   model_name,
   trust_remote_code=True,
   device_map="auto"
)
```

```
# Print all model configuration parameters
config = model.config
print("\n=== Model Configuration Parameters ===")

# Architecture parameters
print("\nArchitecture Parameters:")
# Size of the hidden layers
print(f"Hidden size: {config.hidden_size}")
# Number of transformer blocks
print(f"Number of layers: {config.num_hidden_layers}")
# Number of attention heads
print(f"Number of attention heads: {config.num_attention_heads}")
# Size of the MLP intermediate layer
print(f"Intermediate size: {config.intermediate_size}")
# Tokenizer parameters
print("\nTokenizer Parameters:")
# Size of the vocabulary
print(f"Vocabulary size: {config.vocab_size}")
# Maximum sequence length
print(f"Maximum position embeddings:{config.max_position_embeddings}")

# Model-specific parameters
print("\nModel-specific Parameters:")
for key, value in config.to_dict().items():
   if key not in ['architectures', 'model_type', 'torch_dtype']:
       print(f"{key}: {value}")
```

Before working with a model—especially an LLM—it's a common and recommended practice to inspect its configuration. A model's configuration contains valuable details about its architecture, such as the number of layers, hidden dimensions, attention heads, vocabulary size, and more. Understanding this information helps you estimate the compute resources required (such as GPU memory), choose appropriate model serving strategies (like quantization or batching), and plan for performance optimization (for instance, layer-wise parallelism or model sharding).

Here is Qwen 2.5's model configuration:

```
>> Architecture Parameters:
Hidden size: 896
Number of layers: 24
Number of attention heads: 14
Intermediate size: 4864

>> Tokenizer Parameters:
Vocabulary size: 151936
Maximum position embeddings: 32768

>> Model Size:
Total parameters: 494,032,768

>> Model-specific Parameters:
```

```
vocab_size: 151936
max_position_embeddings: 32768
 .. .. ..
```

By inspecting the configuration up front, you can make better decisions about deployment strategies—like whether to use quantized models, distillation, multi-GPU setups, or even offloading techniques.

Transformer (decoder) block

Next, let's zoom into the Transformer blocks. Decoder blocks are the most critical and computation-heavy layers in the LLM architecture (Figure 2-4).

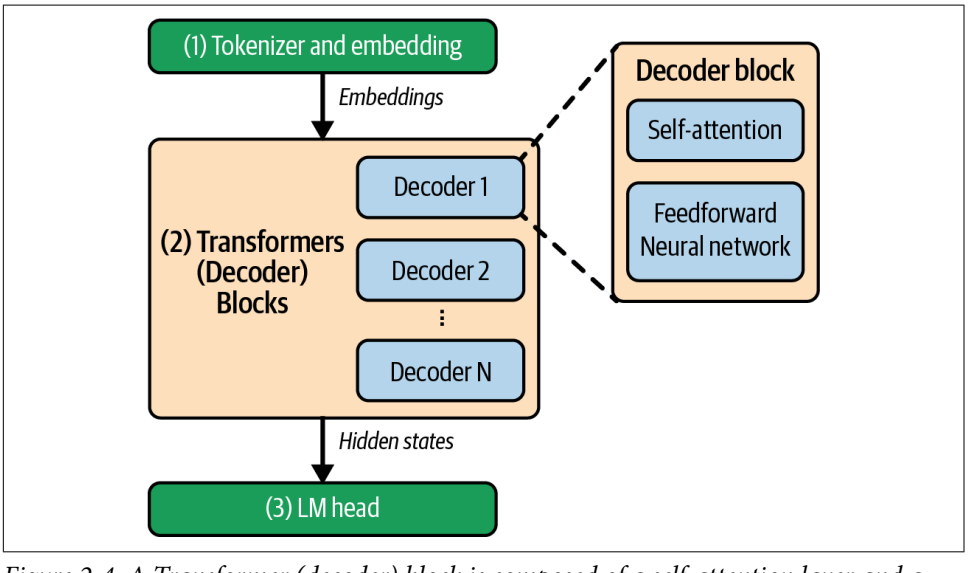

Figure 2-4. A Transformer (decoder) block is composed of a self-attention layer and a feedforward layer

In Figure 2-4, you can see that a Transformer block contains two primary components:

Self-attention layer
: This is the novel component of Transformer models, which incorporate *contextual information* by dynamically correlating input tokens based on their meaning and relative positions. Calculating *attention* (for context information) is crucial in LLMs, since context allows models to understand the meaning of words and sentences based on surrounding information, reducing ambiguity and enabling coherent interpretation.

  In simple terms, the self-attention layer allows the model to consider all previous tokens in the prompt and assign different levels of importance to each when

generating the next word. In the next section, we'll take a closer look at how attention works.

Feedforward neural network (FFN)
: FFN refines the understanding of a given token into a stronger representation: the contextual meaning of a token, usually a dense, high-dimensional vector (for example, 768 or 4,096 dimensions). This component applies per-token transformation. It leverages the context information from the attention layer, incorporates all the knowledge it learned during training, and provides a refined token representation before making a prediction.

The following code shows how to inspect the decoder layer of the Qwen 2.5 model:

```
print("\nModel Structure:")
def print_module_structure(module, prefix=''):
   for name, child in module.named_children():
       # Skip certain internal modules for clarity
       if name in ['_orig_mod', 'wrapped_model']:
           continue

       # Print the current module
       print(f"{prefix}{name}: {type(child).__name__}")

       if "Qwen2Attention" in name.lower():
         print(f"\nFound attention module: {name}")
         print(f"Type: {type(module).__name__}")

         # Print attention-specific attributes
         if hasattr(module, 'num_heads'):
             print(f"Number of attention heads: {module.num_heads}")
         if hasattr(module, 'head_dim'):
             print(f"Head dimension: {module.head_dim}")
         if hasattr(module, 'hidden_size'):
             print(f"Hidden size: {module.hidden_size}")
         if hasattr(module, 'rotary_emb'):
             print(f"Has rotary embeddings: {module.rotary_emb is not None}")
```

With the code, you can clearly see the structure of the decoder layer, including `attention:Qwen2Attention`, `mlp:Qwen2MLP`, and `layernorm:Qwen2RMSNorm`:

```
Model Structure:
model: Qwen2Model
  embed_tokens: Embedding
  layers: ModuleList
    0: Qwen2DecoderLayer
      self_attn: Qwen2Attention
        q_proj: Linear
        k_proj: Linear
        v_proj: Linear
        o_proj: Linear
      mlp: Qwen2MLP
```

```
      gate_proj: Linear
      up_proj: Linear
      down_proj: Linear
      act_fn: SiLU
    input_layernorm: Qwen2RMSNorm
    post_attention_layernorm: Qwen2RMSNorm
  1: Qwen2DecoderLayer
    self_attn: Qwen2Attention
  .. .. ..
```

Knowing this decoder layer structure is crucial when planning model optimization or serving. For example, attention-heavy blocks may benefit from fused attention kernels or custom CUDA implementations, which we will discuss in the following chapters.

Capture Token Context by Calculating Attention

Context is critical for language processing tasks. For example, given the sentence "I saw a dog chasing a squirrel, and it climbed up the tree" without context, it's unclear whether *it* refers to the dog or the squirrel.

Ever since the 2017 publication of "Attention Is All You Need", self-attention has become the main attention mechanism through which Transformers allow each token in a sequence to "look at" other tokens and weigh their relevance when computing its own representation.

For example, when an LLM processes the prompt "Write a short introduction about the US capital city," at the token capital, the self-attention allows the model to:

- Look back at `US` to understand that `capital` is referring to a country's capital, not financial capital.
- Consider `introduction` as a direction to keep the writing concise and general.
- Use `Write` to know the overall task is instructional.

Attention calculation

In a nutshell: for each token, the LLM computes three vectors: query (Q), key (K), and value (V). The attention score of a token is calculated by taking the dot product between the query of the token and the keys of all the tokens of the input sequence, scaling the result, applying a softmax to get weights, and then using those weights to compute a weighted sum of the value vectors. This result becomes the updated representation of the token, enriched with context from the entire sequence. The attention mechanism is summarized by the following equation:

$$\text{Attention}\left(Q, K, V\right) = \text{softmax}\left(\frac{QK^t}{\sqrt{d_k}}\right)V$$

For a detailed explanation of this calculation, please refer to section 3.2.1, "Scaled Dot-Product Attention," in "Attention Is All You Need".

Multi-head attention

To further enhance the model's understanding, instead of doing one attention calculation per token, the model performs multiple attention calculations for the same token, called *heads*, each with its own Q/K/V projections. This allows the model to capture different types of relationships (for example, syntactic, positional, and semantic) in parallel. The outputs from all heads are then concatenated and passed through a linear layer to produce the final attention output. Figure 2-5 illustrates a highly abstracted view of the multi-head self-attention mechanism described here.

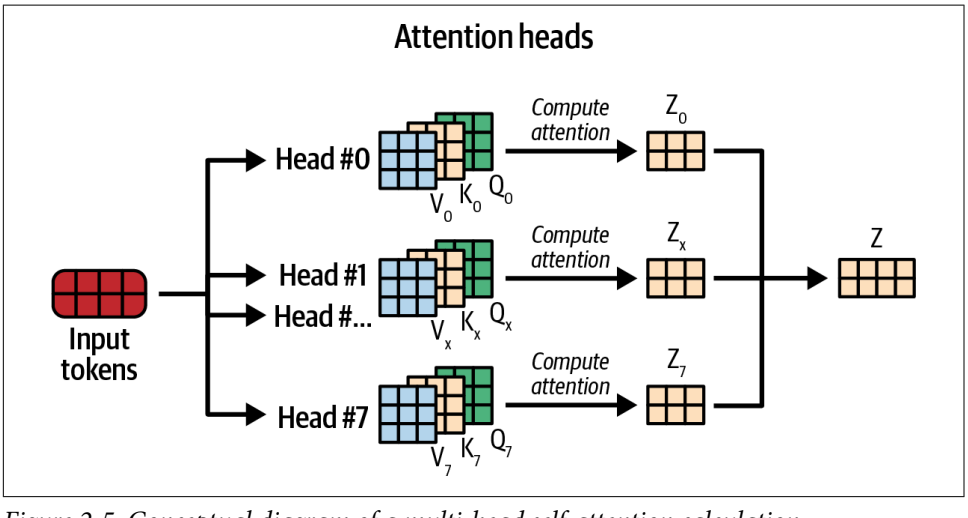

Figure 2-5. Conceptual diagram of a multi-head self-attention calculation

Now, let's see how attention works in practice. In the following code, for the input sequence "write a short introduction about the US capital city," we use the BertViz library's `head_view` class to visualize the attention of each decoder layer of the Qwen model. You can try this code yourself to look at how different attention heads in a layer connect different tokens:

```
text = "write a short introduction about the US capital city"
model_name = "Qwen/Qwen2.5-0.5B"
tokenizer = AutoTokenizer.from_pretrained(model_name, output_attentions=True)
model = AutoModelForCausalLM.from_pretrained(
    model_name, output_attentions=True
).eval().cuda()
```

```
# Tokenize input
inputs = tokenizer(text, return_tensors="pt").to(model.device)

# Generate outputs with attention
with torch.no_grad():
   outputs = model(**inputs, output_attentions=True)

# Get attention weights
attention = outputs.attentions

# Use bertviz and attention weights to visualize
# the relations between the input tokens
tokens = tokenizer.convert_ids_to_tokens(inputs['input_ids'][0])
head_view(attention, tokens)
```

By specifying `output_attentions=True`, we could fetch the attention weight of the given model input, "write a short introduction about the US capital city," after token generation. And then we could visualize tokens relations in each decoder layer by `head_view(attention, tokens)`. See the output graph in Figure 2-6.

Each line in Figure 2-6 represents an attention connection between two tokens—the line thickness indicates the strength of the attention weight, while color distinguishes different attention heads. By examining these connections, we can observe which tokens a given token attends to most strongly. In this example, the token `capital` shows strong attention toward `US` and `write`, suggesting these tokens are influential in shaping its contextual representation at layer 10.

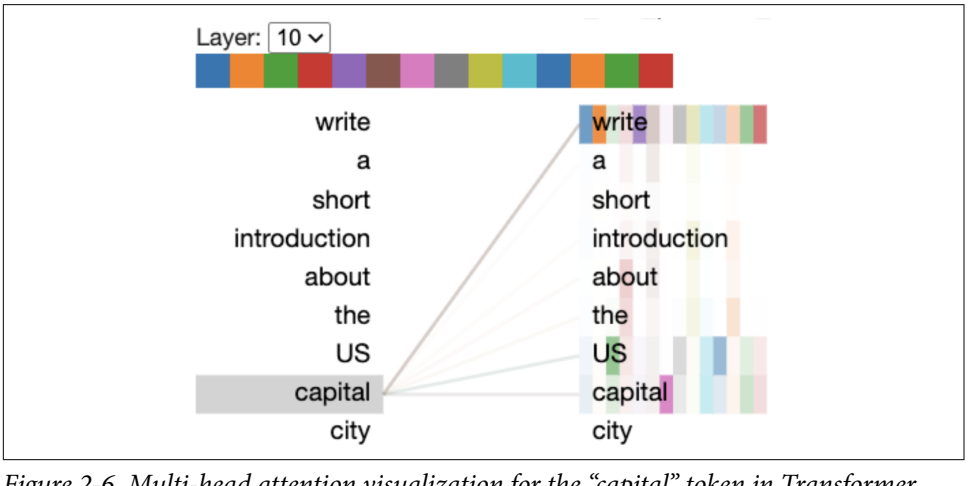

Figure 2-6. Multi-head attention visualization for the "capital" token in Transformer layer 10

It's important to note that the attention computation described here is intentionally high level and conceptual. We've abstracted away many implementation details—such

as token and embedding dimensions, matrix shapes within each attention head, and the exact mechanics of output aggregation. This simplification is deliberate: for ML engineers focused on model serving and optimization, the current level of understanding—centered on how attention works and how tokens are generated in a Transformer—is typically both sufficient and practical.

However, if you are interested in exploring the attention mechanism in greater depth, I highly recommend "The Illustrated Transformer" blog, which walks through attention calculations in an intuitive, non-mathematical way, as well as books such as *Hands-On Large Language Models* and *Build a Large Language Model (From Scratch)*, which offer detailed and comprehensive explanations of transformers and attention.

You Just Need to Grasp the Transformer Concept—Not the Math

Many people find the mathematical details of attention and Transformers intimidating—especially the matrix-heavy computations often emphasized in academic papers. However, for ML engineers focused on LLM serving, it's both practical and—often—preferable to understand the attention mechanism conceptually, without diving deep into the math.

While a detailed understanding of self-attention can be valuable, it is not a prerequisite for effectively serving or optimizing LLMs. From a serving perspective, what's most important to know is that attention is compute-intensive, especially during the prefill phase (which we will discuss shortly), and that its memory and latency costs scale with input sequence length. This level of understanding is critical when you're tuning inference performance, implementing KV caching, or distributing workloads across GPUs.

Now that you've built a solid understanding of LLM architecture and the autoregressive generation process, let's shift to a more practical perspective. In the next section, we'll walk through how to execute LLM generation step by step, using hands-on code examples to illustrate each stage of the process.

Executing LLM Generation: A Step-by-Step Walkthrough

In this section, we'll demonstrate the core internal LLM serving concepts, such as KV cache, prefill, and decode, by showing you behind the scenes of how a LLM generates tokens—step by step.

Our goal is not to teach Transformer theory but to build a serving-oriented mental model of how an LLM executes during inference—because this execution pattern is what drives almost every design decision in LLM serving systems.

Please note that in our demo code, we'll use a single-request and single-sequence example for demonstration purposes; the code example is not recommended in production settings. Later chapters will show how serving systems batch, stream, and optimize this same execution pattern at scale.

As we walk through LLM generation line by line, pay attention to how execution unfolds over time, not just what each line of code does. Concepts introduced here—such as token-level execution, prefill versus decode, and cached state—will reappear throughout the book when we discuss batching, streaming, KV cache optimization, and serving frameworks like vLLM and SGLang.

By the end of this walkthrough, you'll have a clear and practical understanding of how LLM inference works. If some details feel unfamiliar on the first pass, that's OK. The purpose of this section is to give you intuition; we'll revisit and operationalize these ideas in later chapters.

Run the Qwen Model

To get started easily, let's run the Qwen 2.5 model using the Hugging Face pipeline library. This is the simplest way to use an LLM, as the `pipeline` object abstracts away much of the underlying complexity and provides a straightforward `generator()` API for text generation:

```
# Initialize the text generation pipeline
generator = pipeline('text-generation', model='Qwen/Qwen2.5-0.5B')
# Define your prompt
prompt = "Write a short introduction about the US capital city."
# Generate text
generated_text = generator(prompt, max_length=50, num_return_sequences=1)
# Print the generated text
print(generated_text[0]['generated_text'])
```

This code loads the Qwen 2.5 model and generates text using `generator(prompt, ...)`. For the input prompt "Write a short introduction about the US capital city," the output is:

```
Write a short introduction about the US capital city.
The United States of America is the largest country in
the world by area, .. .. ..The city is also home to many
other important institutions, including the National Mall,
the White House Museum, and the National Archives.
```

Model Prediction, Line by Line

Now, let's take a look at what happens under the hood of the `generator()` function. Before diving into the code, let's first revisit the token-by-token generation workflow, illustrated in Figure 2-7.

Figure 2-7 highlights two key aspects of LLM generation. First, the LLM generates one token at a time; second, each newly generated token is appended to the previous input sequence to form the new input for the next prediction step.

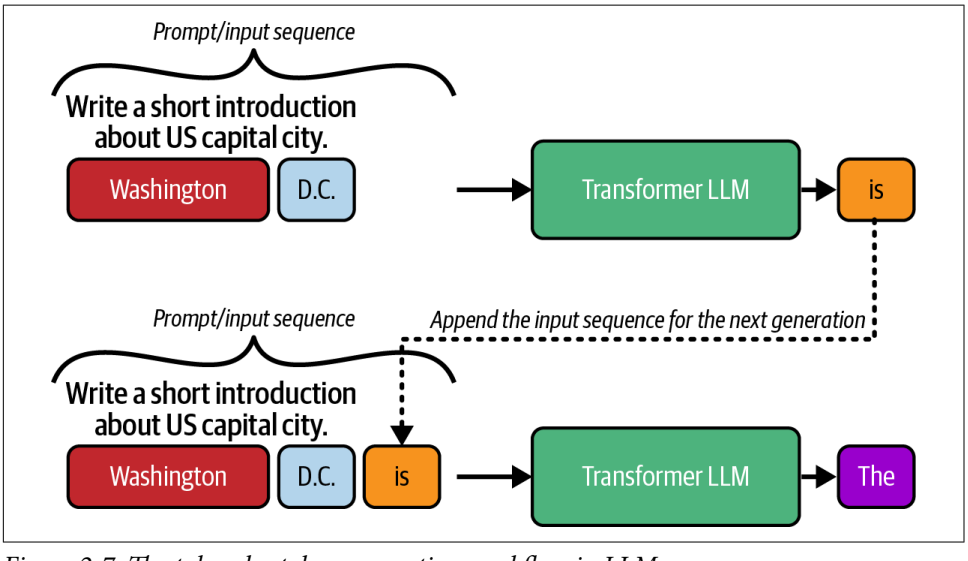

Figure 2-7. The token-by-token generation workflow in LLMs

Now, let's examine how this process is implemented. First, we need to specify the model and load tokenizer. In Example 2-1, instead of using pipeline, we use `AutoModelForCausalLM` to load the Qwen 2.5 model. It offers full control over the generation configuration, inputs, and decoding strategies—making it the ideal choice when we want to manually prepare inputs and handle the generation process step by step.

Example 2-1. Implementing Qwen token generation, token by token

```
# (1) Load tokenizer and model
model_name = "Qwen/Qwen2.5-0.5B"
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
model = (
    AutoModelForCausalLM.from_pretrained(
        model_name, trust_remote_code=True,
    ).to("cuda" if torch.cuda.is_available() else "cpu")
)
```

Next, define the input prompt:

```
# (2) Define the input prompt
prompt = """The history of human … … … How might the next wave of
communication tools shape our relationships, societies, and sense
of identity?"""
```

Now, it's time to tokenize the prompt—that is, convert it to an input format that the model understands. `idx = tokenizer(prompt,..)` tokenizes the prompt as the initial model input:

```
# (3) Convert (tokenize) prompt to the input format that model understands
max_new_tokens = 100
# tokenize the input prompt for the first output token
# PS: prompt is the initial input sequence for LLM generation
idx = tokenizer(prompt, return_tensors="pt").input_ids.to(model.device)
start_time = total_time = time.time()
times = []
```

Then we start the main generation loop, generating tokens one by one. Notice that `outputs = model(idx_cond)`, at step 4(B), runs the Qwen model and produces candidates for the new token, given the input `idx_cond`, while `logits = outputs.logits` gets the *logits*, or raw prediction scores, for each token prediction:

```
# (4) Main generation loop - generate tokens one by one
for _ in range(max_new_tokens):
   # (A) Set the current context for generation
   idx_cond = idx
   with torch.no_grad():
       # (B) Generate predictions (token candidates) for next token
       outputs = model(idx_cond)
       # Get the logits (raw prediction scores) for each token prediction
       logits = outputs.logits
```

Next, the model selects the next token from the predictions it just generated:

```
# (C) Select next token from the predictions generated in step (B)
   logits = logits[:, -1, :] #  Select only the logits for the last token
   # Convert logits to probabilities using softmax
   probas = torch.softmax(logits, dim=-1)
   # Sample the next token from the probability distribution of the
   predicted tokens from step (B)
   idx_next = torch.multinomial(probas, num_samples=1)
   print("Next Token is:", tokenizer.decode(idx_next[0])
   time_cost = time.time() - start_time
   times.append(time_cost)
```

Append the new token to the input sequence using `idx = torch.cat((idx, idx_next), dim=1)`. Finally, check that an end-of-sequence token was generated; `idx` also holds the final output text:

```
# (D) Append the new token to the input sequence
idx = torch.cat((idx, idx_next), dim=1)

# (E) Check if end-of-sequence token was generated
if idx_next.item() == tokenizer.eos_token_id:
    print("\n[Generation completed - EOS token reached]")
    break
```

Finish by decoding the entire generated sequence:

```
# (5) Decode the entire generated sequence
generated_text = tokenizer.decode(idx[0], skip_special_tokens=True)
```

You can try out this code yourself in the *ch2_Workthrough_LLM_execution.ipynb* notebook. In addition to demonstrating the code, we also tracked the latency of each token during the generation process. The demo generated 100 tokens in 9.1205 seconds—averaging approximately 0.09 seconds per token. See Figure 2-8 for details.

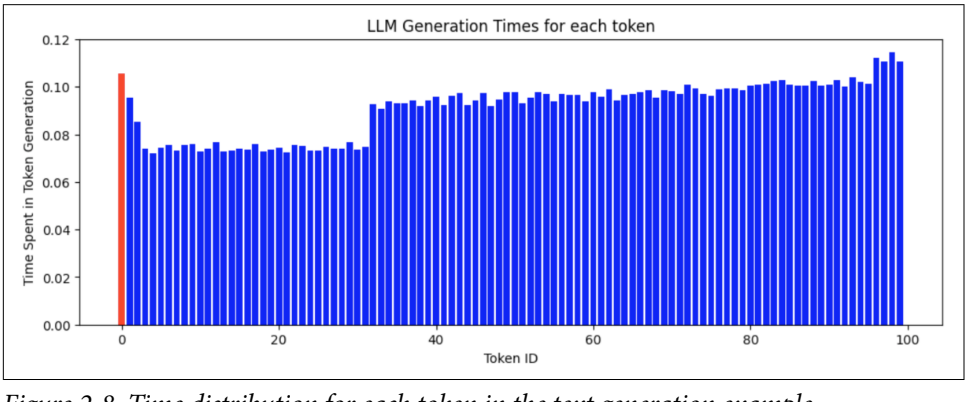

Figure 2-8. Time distribution for each token in the text generation example

You may also notice a pattern in Figure 2-8: except for the first token, the per-token generation time tends to gradually increase. This is expected—since each newly generated token is appended to the input sequence, the model must process a longer context with every step. As a result, both computation time and resource usage grow with each token.

As machine learning engineers, we can quickly recognize the inefficiency shown in Figure 2-9: the model recomputes attention for all previous tokens every time a new token is generated. This repetitive computation over known tokens is costly and unnecessary. Can we avoid recalculating the same attention for previous tokens and instead focus computation on the newly generated one?

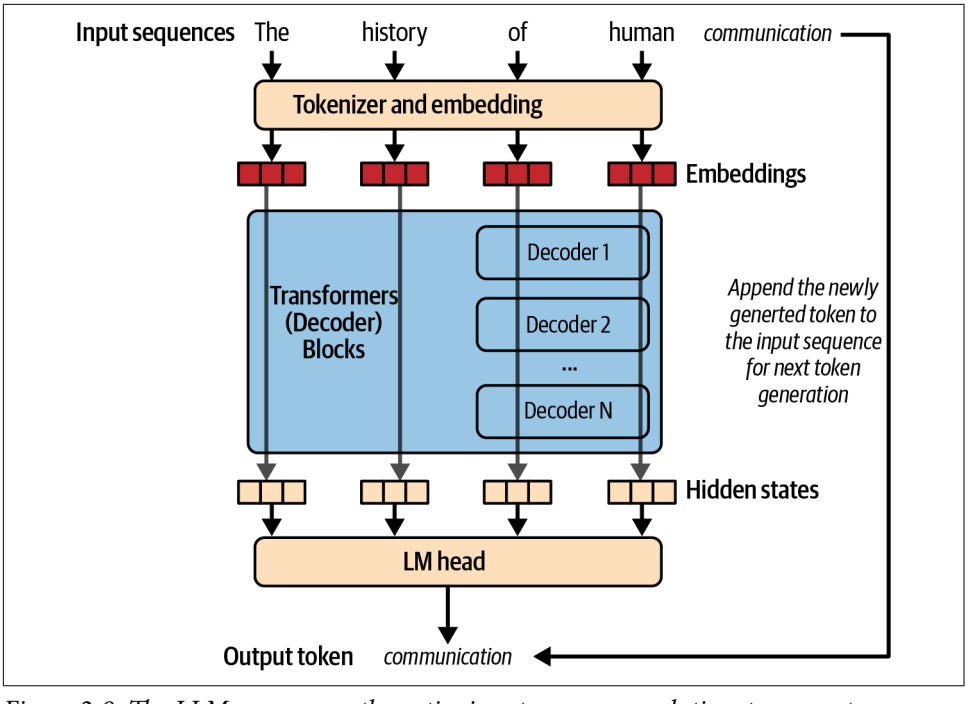

Figure 2-9. The LLM reprocesses the entire input sequence each time to generate a new token, leading to increased computation as the sequence grows

This brings us to the next topic—the concept of KV cache.

Enable the KV Cache to Boost Performance

Now, imagine if you could cache the intermediate results from all previous tokens—specifically, the key and value vectors computed by the attention mechanism. Then, during each new token generation step, you would only need to process the newly added token, rather than recomputing attention for the entire sequence. That's what enabling the KV cache does: this optimization stores the attention keys and values computed at each layer for previously generated tokens, allowing the model to skip redundant computations during decoding (see Figure 2-10).

As shown in Figure 2-10, the KV cache shifts the LLM calculation from full-sequence recomputation to an incremental, cache-augmented workflow—trading increased memory usage for significant compute savings. This technique dramatically reduces redundant computation and enables much faster token generation.

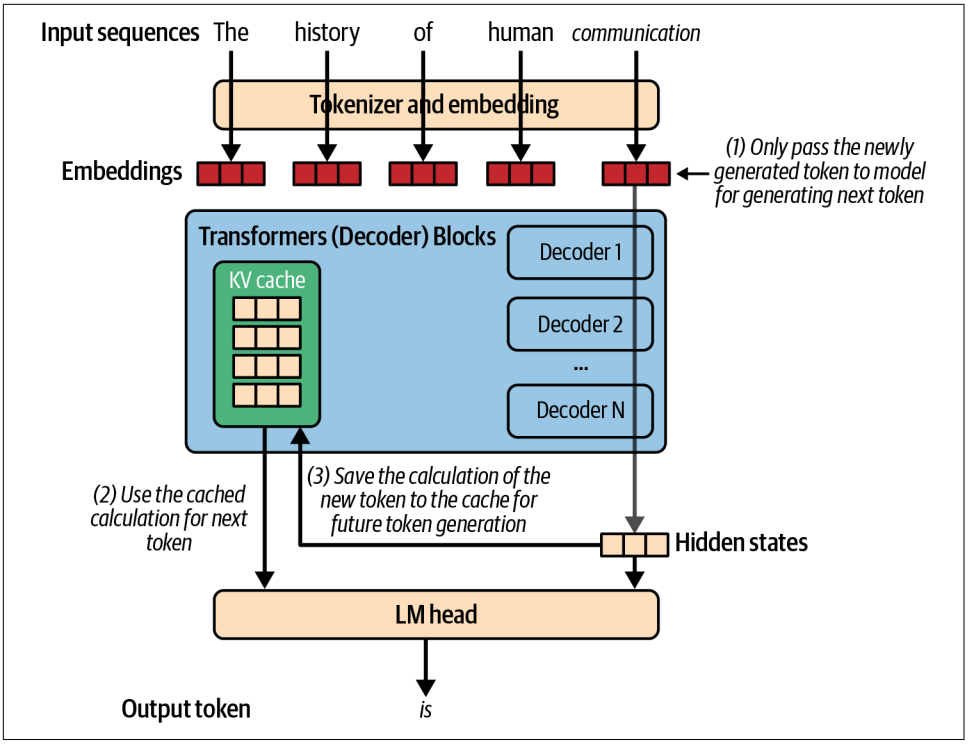

Figure 2-10. Accelerating LLM generation with KV caching: only the new token is processed while previously computed key-value pairs are reused, reducing redundant computation

With this understanding in mind, let's look at how KV caching can be used with the code in Example 2-2.

Example 2-2. Implementing Qwen text generation, token by token, with KV caching enabled

```
# (1) Define key/value cache for faster generation
past_key_values = None

for _ in range(num_interations):
   print("input_ids size: " + str(input_ids.size()))
   with torch.no_grad():
       outputs = model(input_ids=input_ids,
         # (2) Use KV-cache from previous iteration
         past_key_values=past_key_values,
         # (2) Enable KV caching
         use_cache=True,
         max_new_tokens = 100,
         min_new_tokens= 100)
```

```
        logits = outputs.logits
        # (3) Update KV Cache
        past_key_values = outputs.past_key_values
        torch.cuda.synchronize()

    logits = logits[:, -1, :]
    probas = torch.softmax(logits, dim=-1)
    generated_token_id = torch.multinomial(probas, num_samples=1)

    # (4) Update input_ids with only the new token (using KV-cache)
    # Note: Not concatenating with previous tokens due to KV-cache
    input_ids = generated_token_id
    idx = torch.cat((idx, generated_token_id), dim=1)

    if generated_token_id.item() == tokenizer.eos_token_id:
        print("\n[Generation completed - EOS token reached]")
        break
```

The biggest differences between Examples 2-1 and 2-2 are:

- Example 2-2 only uses the newly generated token, `generated_token_id`, as the model generation input for the next token: `input_ids = generated_token_id` and `outputs = model(input_ids=input_ids, .. )`.
- In Example 2-2, besides passing in the token, we also specify the KV cache as the model generation input, see `outputs = model(input_ids=input_ids,past_key_values=past_key_values, ..)` at step 2.
- Step 3 of Example 2-2 updates the KV cache after each token generation with `past_key_values = outputs.past_key_values`.

Our total execution time for Example 2-2 was 3.1416 seconds. Figure 2-11 compares the execution times over 100 tokens of Example 2-1 (left, no cache) and Example 2-2 (with cache).

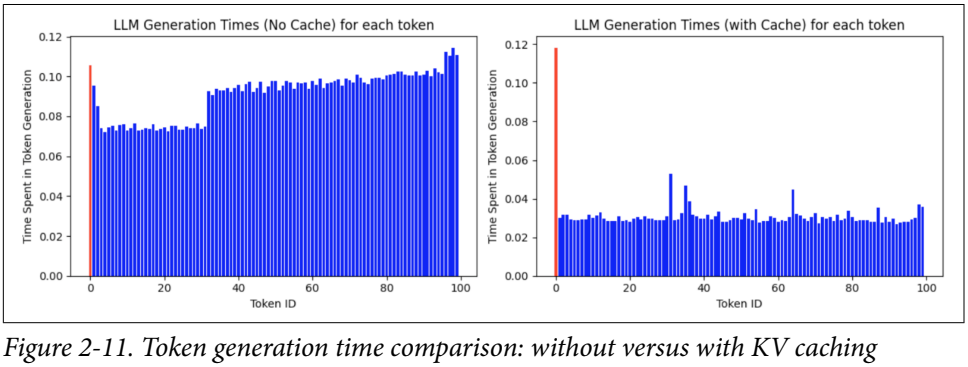

Figure 2-11. Token generation time comparison: without versus with KV caching enabled

Without caching, the generation time increases steadily as the model reprocesses the growing input sequence at each step. In contrast, enabling the KV cache significantly reduces compute time by reusing previously computed key-value pairs—resulting in more stable and efficient generation speed after the first token.

With this example, you should now have a clear, hands-on understanding of how effective the KV cache is in improving LLM generation performance—both in terms of latency and throughput.

In practice, many advanced techniques have been proposed to design, manage, and optimize KV caches more efficiently. With the foundational intuition you've gained here, you'll be better equipped to explore those methods and apply them in your own work. We'll also cover KV cache optimization strategies in more detail in Chapters 6 and 7, focused on performance tuning.

The Prefill and Decode Phases

Now that you've seen how tokens are generated, you're ready to explore the next two phases, *prefill* and *decode*. These terms are widely used in the LLM serving domain, particularly in the context of performance optimization and scalability. Figure 2-12 illustrates the concept.

Let's look at these phases one by one:

Prefill phase
: In the prefill phase, also called *prompt processing*, the model processes the entire input prompt at once. This phase is *compute-intensive* because attention is computed across all tokens in the prompt (quadratic complexity with respect to sequence length).

Decoding phase
: In the decoding phase, also called *token-by-token generation*, the model generates one token at a time and repeats for every new token. The sequence grows, but the (attention) computation typically focuses only on the most recent token.

As Figure 2-12 shows, the prefill phase occurs when the user submits the prompt "Write a short introduction about the US capital city." During this phase, the model processes all tokens in the prompt simultaneously, leveraging parallelism for relatively efficient computation.

Once the prefill phase is complete, the decoding phase begins. This is where the model generates tokens one at a time to form a response, such as `Washington, D.C.`, `is`, and `the`.

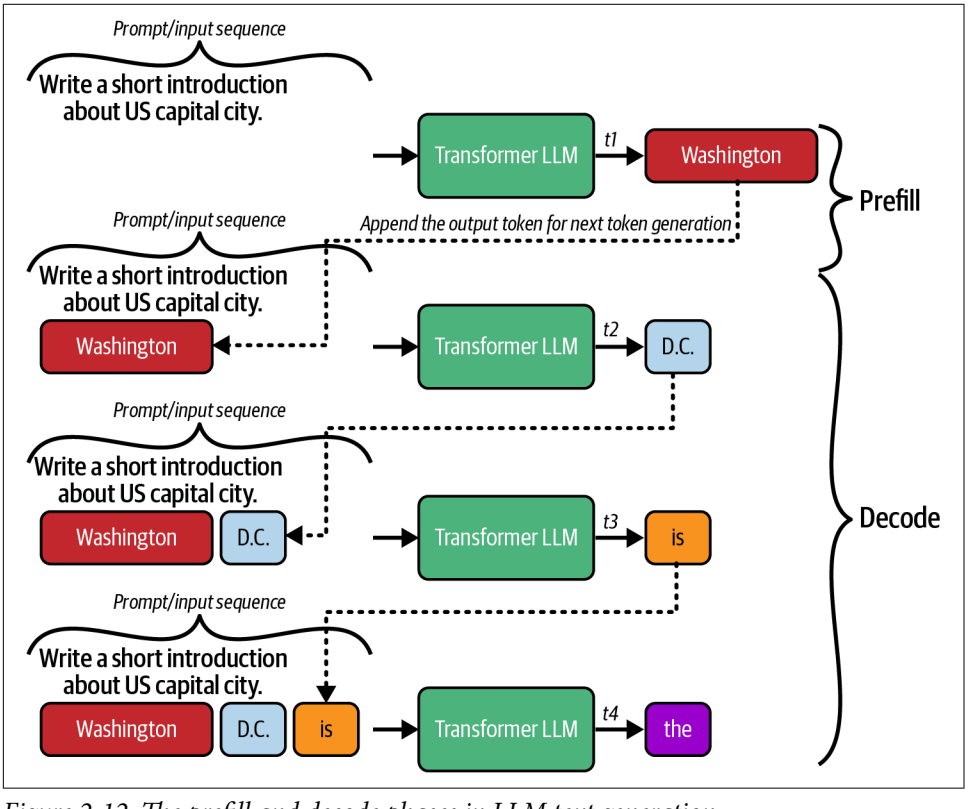

Figure 2-12. The prefill and decode phases in LLM text generation

Figure 2-13 compares the difference in processing time between the prefill and decoding phases when KV caching is enabled. The first bar on the left represents the time taken to generate the first token. This corresponds to the prefill phase, which is significantly more time-consuming because the model must process the entire prompt at once. In contrast, the following bars (representing the decoding phase) show much shorter and more consistent token-generation times, as the model generates one token at a time using the cached information from earlier steps.

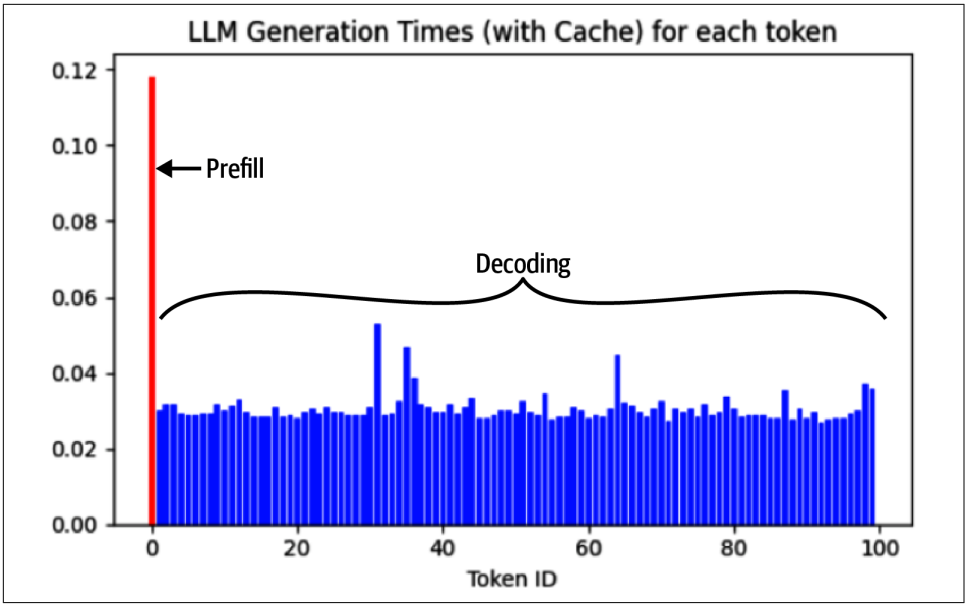

Figure 2-13. Token generation time in the prefill and decoding phases of Example 2-2 (with KV cache enabled)

Why Does Learning About the Prefill and Decoding Phases Matter?

Understanding the difference between prefill and decoding phases—and their respective compute patterns—is critical for making informed optimization and resource-planning decisions in LLM serving. The prefill phase is compute-intensive due to parallel processing multiple prompt tokens; the decoding phase is memory-intensive, primarily due to the frequent loading of model weights and the growing size of the KV cache. Knowing which phase dominates in your use case helps you target the right bottlenecks.

For example, in scenarios involving long prompts, such as processing a 500+ page PDF, the prefill phase becomes expensive. On the other hand, if you're working with short prompts and long generations—such as in chatbot replies or story generation—the decoding phase becomes the bottleneck.

By distinguishing between these phases, you can better align your system design and optimization efforts with the performance characteristics of your specific application. We will discuss phase-specific optimization techniques in Chapters 6 and 7.

In this section, we demonstrated how to implement inference from scratch—manually loading the model and generating tokens step by step. While this approach provides valuable insight into how LLMs work under the hood, real-world

applications typically rely on serving frameworks rather than custom-built inference pipelines. In the remainder of this chapter, we'll use the vLLM serving framework as a practical example to show you how to execute LLM inference with production-ready libraries.

Run the LLM with a Serving Framework

Model serving frameworks—such as vLLM and SGLang—are purpose-built systems designed to load pre-trained language models and expose them through APIs (typically REST or gRPC) for real-time or batch inference. While training frameworks like PyTorch and TensorFlow focus on model development and gradient-based learning, serving frameworks are optimized for efficient, scalable, and low-latency inference.

But serving frameworks offer far more than just the ability to run inference. They provide essential capabilities such as:

- Efficient decoding with KV-cache reuse
- Request scheduling (for example, batching or micro-batching)
- Support for multi-user concurrency
- Token streaming, cancellation, and interruption handling

A well-designed serving framework (like vLLM) also continually integrates the latest research optimizations—such as paged attention and speculative decoding—while abstracting away low-level infrastructure concerns. This allows developers to focus on building applications rather than re-implementing inference logic for each model architecture or chasing the latest academic papers about improvements.

In this section, we'll walk you through how to use vLLM to load and serve an LLM model. You can compare this with the manual inference implementation we explored in the previous section to appreciate the practical advantages of using a serving framework.

Serve the LLM (Qwen) with vLLM

Serving an LLM with vLLM is quite simple and efficient. With just a few lines of code, you can load a model such as Qwen 2.5 and run inference. Here's a basic example:

```
import time
from vllm import LLM, SamplingParams

model_name = "Qwen/Qwen2.5-0.5B"

# Load model with vLLM
llm = LLM(model=model_name, dtype="float16")

# Define the prompt.
```

```
prompt = """You are an expert AI historian writing a
detailed chapter for a book titled "The Evolution of
Human-AI Collaboration." … … Write in a formal tone,
with rich detail and examples in each era."""

# Create inference parameters.
inference_params = SamplingParams(temperature=0.8, top_p=0.95, max_tokens=128)

# Run token (text) generation with prompt and inference parameters.
outputs = llm.generate([prompt], inference_params)

# Print the results.
for output in outputs:
 print(f"Generated text: {output}")
```

As you can see, the `LLM()` and `generate()` functions abstract away much of the complexity, enabling quick experimentation with LLMs. Despite its simplicity, vLLM provides extensive configuration options to fine-tune performance and behavior to meet your specific application needs. Here's an example with more advanced configuration:

```
# Configure Model Loading
model = LLM(
   model="Qwen/Qwen-7B",
   # Memory management
   swap_space=16,  # CPU swap space size in GB
   max_model_len=4096,  # Maximum model length
   # PagedAttention settings
   block_size=16,  # Block size for PagedAttention
   enable_prefix_caching=True,  # Enable prefix caching
   # KV cache management
   max_num_sequences=256,  # Maximum number of sequences
   max_sequence_length=4096,  # Maximum sequence length
   # Performance optimizations
   enable_chunked_prefill=True,  # Enable chunked prefill
   enable_cuda_graph=True,  # Enable CUDA graph
   # System settings
   worker_use_ray=False,  # Use Ray for distributed serving
   disable_custom_all_reduce=False,  # Disable custom all-reduce
)

# Configure Model Inference Parameters
sampling_params = SamplingParams(
   temperature=0.7,  # Controls randomness (0.0 = deterministic)
   top_p=0.9,  # Nucleus sampling parameter
   top_k=50,  # Top-k sampling parameter
   max_tokens=100,  # Maximum tokens to generate
   stop=["\n", "###"],  # Stop sequences
   frequency_penalty=0.1,  # Frequency penalty
   presence_penalty=0.1,  # Presence penalty
   repetition_penalty=1.1,  # Repetition penalty
```

```
    skip_special_tokens=True,  # Skip special tokens in output
)
```

vLLM follows the parameter conventions of the OpenAI text completion API, making it familiar and easy to adopt. You can find a full list of model-loading configuration options in the vLLM GitHub repository.

Performance Comparison: vLLM Versus Hugging Face Transformers

In addition to convenience, vLLM delivers substantial performance improvements. In many benchmarks, it achieves 10 to 20 times higher throughput than the Hugging Face `.generate()` API (Transformer library). Let's run a quick side-by-side comparison using the same prompt and model. Here is the Hugging Face code:

```
# Run model inference with Hugging Face library
# Load model and tokenizer in Hugging Face library
tokenizer = AutoTokenizer.from_pretrained(
    "Qwen/Qwen2.5-0.5B",
    trust_remote_code=True,
)
model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2.5-0.5B",
    device_map="auto",
    trust_remote_code=True,
)

start_time_basic = time.time()
# Create the model prediction pipeline
generator = pipeline('text-generation', model=model, tokenizer=tokenizer)
# Generate prediction with prompt
outputs_basic = generator(prompt, max_length=128, temperature=0.8, top_p=0.95)
end_time_basic = time.time()
```

vLLM takes 1.12 seconds, while the Hugging Face library takes 19.58 seconds. That's a 17x speedup on a single prompt—and the performance gap only widens when you scale up to concurrent or batched inference.

Best Practice: Start Simple, Then Optimize

In real-world development, teams can prototype with Hugging Face Transformers for ease of use, then migrate to frameworks like vLLM for production usage and fine-tune the serving configuration for better latency, throughput, and concurrency. By allowing you to tune its serving parameters to fit your specific model and use case, frameworks like vLLM offer enterprise-grade performance with minimal engineering overhead.

LLM Streaming Serving Basics

In our vLLM code example, you might have noticed that although the LLM generates tokens one by one internally, the call to `outputs = llm.generate([prompt], inference_params)` waits until the entire output is generated before returning any result. In the context of a web service—for example, a chatbot—this means the user experiences a long delay before receiving any response—from a few seconds to even minutes, depending on the output length.

For applications with conversational interfaces (like chatbots) or those requiring real-time feedback (such as live captioning or summarization), this delay significantly impacts user engagement. A slow response can frustrate users, reduce retention, and ultimately, lead to a failure to meet business goals.

As you learned in the previous section, tokens are generated sequentially during the decoding phase. To improve responsiveness, you can return each token immediately as it is generated—a technique known as *LLM streaming*. *Streaming*, in LLMs, refers to the process of incrementally returning output token by token (or in small chunks) during generation, rather than waiting for the entire output to complete.

Let's look at how to enable streaming generation with vLLM using a simple code example:

```
# Initialize the vLLM async streaming engine arguments
engine_args = AsyncEngineArgs(
    model="Qwen/Qwen2.5-0.5B",
    dtype="float16",
    .. .. ..
)

# Create the vLLM async streaming engine
engine = AsyncLLMEngine.from_engine_args(engine_args)

# Define the async function to generate text in streaming mode
async def generate_text(prompt: str, max_tokens: int = 100):
    try:
        # Define sampling parameters
        sampling_params = SamplingParams(
            temperature=0.0,
            max_tokens=max_tokens,
            stop=["\n"],  # Stop at newline
        )

        # Generate text
        request_id = "test-request"  # Unique ID for this request
        # Generate tokens in streaming mode
        results_generator = engine.generate(
            prompt=prompt,
            sampling_params=sampling_params,
```

```
        request_id=request_id
    )

    # Process the results
    final_output = None
    async for request_output in results_generator:
        final_output = request_output
        # Print each token as it's generated
        for chunk in request_output.outputs:
            print(chunk.text, end="", flush=True)
            # We could yield return the token chunk here
            # if it's a web service
    return final_output
```

The key difference in the streaming version of the vLLM code is that we initialize the model using the `AsyncLLMEngine` class instead of the standard `LLM` class. With `AsyncLLMEngine`, the `generate()` function returns an asynchronous stream object (`AsyncStream`) that allows us to pull newly generated tokens one by one using an `async for loop`—for example, `async for request_output in results_generator`.

Another valuable benefit of streaming is that it allows users to cancel generation midway if the output is heading in an undesired direction. In vLLM, this can be achieved by calling `engine.abort(request_id)`, where `request_id` is a unique identifier associated with the specific generation request. This feature not only improves the user experience by avoiding irrelevant or incorrect completions, it also helps conserve compute resources, which is especially important in production environments where efficiency and cost control are critical.

If you're interested in exploring LLM streaming further, we've included a hands-on example in *streaming.py*. In the next chapter, we'll demonstrate how to implement LLM streaming in a web service setting, enabling real-time interaction in user-facing applications.

LLM Batch Serving Basics

In the examples we've covered so far in this chapter, we've sent one prompt to the LLM at a time. While this approach is simple and effective for prototyping, it becomes a bottleneck in high-throughput scenarios—such as summarizing 100,000 documents, indexing 5,000 PDF files, or serving 20,000 concurrent chatbot users. Processing one input at a time simply doesn't scale.

To address this performance limitation and fully utilize GPU compute resources, you need batching. *Batching* refers to grouping multiple input requests together and processing them simultaneously in a single forward pass through the model. Instead of handling one prompt at a time, batching allows the model to generate outputs for multiple prompts in parallel, significantly improving throughput. While batching can be applied to most ML models, it is especially crucial for LLMs due to their compute

characteristics. Figure 2-14 provides a conceptual illustration of how batching works in LLMs.

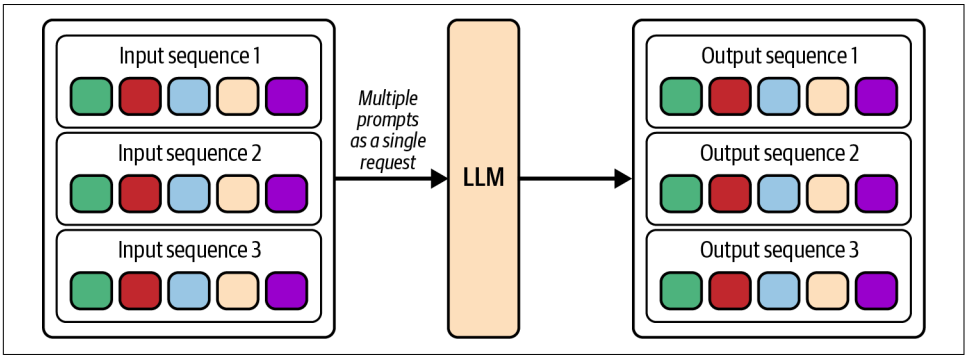

Figure 2-14. Batch LLM inference: processing input sequences and generating output sequences in parallel

Transformer-based LLMs can process input batches efficiently because their computations—like matrix multiplications and attention—can be parallelized across sequences. Thanks to shared model weights and the GPU's parallel compute capability, batching allows handling multiple requests simultaneously with minimal overhead.

We'll explore advanced batching techniques in depth in Chapters 6 and 7. For now, let's walk through a simple experiment to see how batch generation works in practice and compare its latency to single-prompt generation:

```
# Prompts for batch generation, 4 input sequences
prompts = [
    "What is the meaning of life?",
    "Write a short story about a robot learning to love.",
    "Explain quantum physics in simple terms.",
    "Translate 'Hello, world!' into Spanish."
]

sampling_params = SamplingParams(
    temperature=0.8,
    top_p=0.95,
    max_tokens=100
)

start_time = time.time()
# process four input sequences together in one batch
vllm_outputs = llm.generate(prompts, sampling_params)
end_time = time.time()
vllm_time = end_time - start_time

print(
```

```
    f"\nvLLM generation time for 4 prompts in a batch: "
    f"{vllm_time:.4f} seconds"
)

# process prompt one by one
start_time = time.time()
for prompt in prompts:
   vllm_outputs = llm.generate([prompt], sampling_params)
end_time = time.time()
vllm_time = end_time - start_time

print(
    f"\nvLLM generation time for 4 prompts one by one: "
    f"{vllm_time:.4f} seconds"
)
```

When we ran this code experiment, vLLM took 1.0626 seconds to process a batch of four prompts; it took 2.3865 seconds in total to process all four prompts one by one. This demonstrates a 2.2x improvement in throughput when using batching over single-prompt execution.

With further optimization techniques, you can achieve even higher throughput while simultaneously reducing latency. One such technique is *continuous batching* (which we'll discuss in detail in Chapter 6), where new requests are dynamically added as others complete. For example, a 2023 study by Anyscale showed that continuous batching can improve LLM inference throughput by up to 23 times, along with significantly reducing p50 latency.

Summary

In this chapter, we took a deep dive into the core concepts, architectural foundations, and practical techniques for serving LLMs. We started by demystifying the Transformer architecture—focusing on decoder-only models—and examined how attention mechanisms and autoregressive token generation work.

From there, we moved into hands-on walkthroughs that showed how to execute LLM inference. You learned concepts like KV caching and how the prefill and decode phases operate as critical components in optimizing LLM serving performance.

We then transitioned from scratch implementations to using modern serving frameworks like vLLM, demonstrating how they simplify deployment while delivering massive performance. These tools abstract away low-level complexity and enable developers to build scalable, production-ready LLM services with minimal overhead.

Finally, we explored two powerful strategies for real-world LLM application: streaming and batch.

Together, the techniques introduced in this chapter form a foundation on which you can build fast, scalable, and cost-efficient LLM services. You are now equipped with the essential tools and mental models you'll need to reason about LLM performance, troubleshoot bottlenecks, and design more effective serving pipelines.

In the next chapter, we'll build on this foundation and show you how to wrap these serving capabilities into a real-world LLM serving web service, exploring design decisions, API structure, and operational best practices.

CHAPTER 3

Model Serving System Design: A Deep Dive

In Chapter 1, we introduced the major model serving paradigms, outlining common architectural patterns and trade-offs. In Chapter 2, we examined how LLMs perform inference and generate text at the model level. This chapter bridges those foundations to *production engineering*: how to organize code and infrastructure to construct complete serving systems for both single-model and multi-model scenarios.

Model serving is a rapidly evolving field, with hundreds of open source serving frameworks and commercial solutions available. Evaluating, adopting, and customizing the right solution can quickly become overwhelming. Rather than starting with a specific framework, we focus in this chapter on building intuition from first principles. By understanding how serving systems are structured at a fundamental level, you'll be better equipped to reason about any framework or managed service.

To that end, we develop two simplified yet representative serving systems: one for single-model LLM serving and one for multi-model serving. These implementations are intentionally streamlined—they are not meant to replace production frameworks like Triton or vLLM—but they capture the core components and architectural decisions that define real-world systems. Through these examples, you will see how batching, streaming, routing, isolation, and resource management fit together in practice.

We begin by constructing a single-model LLM serving service that supports batching and streaming. From there, we examine a general single-model serving design pattern and its practical constraints. We then extend these ideas to build a multi-model serving system and conclude with an in-depth comparison of two common architectural variations: one optimized for cost efficiency, and another optimized for latency and scalability.

By the end of this chapter, you should be able to reason confidently about what happens under the hood in both single-model and multi-model serving systems.

More importantly, you'll be able to evaluate, adapt, and extend open source or cloud-based serving solutions to match your specific performance, cost, and operational requirements.

Code in the Chapter

You can find the full sample code at this chapter's accompanying GitHub repository. We've selected and simplified key portions of the code for demonstration purposes. For complete implementation details and accurate context, please refer to the full repository. Please also refer to the README file for step-by-step instructions on running the demo services locally.

Build an Online LLM Serving Service from Scratch

Modern serving frameworks such as vLLM and Triton abstract away much of the complexity involved in hosting LLMs. However, those abstractions also hide important architectural trade-offs. To reason effectively about performance, cost, and scalability, it's essential to understand the core mechanics of a serving system before relying on a framework.

In this section, we will build a simplified online single-model LLM serving service. The goal is not to replace production serving frameworks, but to expose the fundamental components they automate: request handling, batching, streaming, scheduling, and resource management for LLM serving.

We will begin with a minimal generation service, then incrementally add batching and streaming capabilities. At the end, we will conclude by showing how vLLM is used to handle batching in practice. By understanding both approaches, you'll be better equipped to reason about single-model serving architectures, evaluate framework choices, and make informed design trade-offs in real-world systems.

Later in the book, particularly in Chapter 8, we will examine vLLM in depth and see how it addresses these same challenges in a highly optimized and scalable way.

Design Goals

In this exercise, you'll build a model serving service that loads a single LLM at startup and supports concurrent generation requests for both batch and streaming. While this example is intentionally simplified to only work for one LLM model that can run on CPU, it covers all the essential components needed to scale into a multi-node, generalized production-grade system with advanced model optimization techniques in the future.

Rather than building a complex, production-ready service, our approach is to implement the core components to help you understand the following critical aspects of LLM serving:

- How web APIs are designed to handle generation requests—including batching and streaming
- What a typical LLM request-processing workflow looks like
- How the service is structured internally to support flexibility and scalability of different LLM models
- How concurrent requests are grouped and processed in batches
- How streaming generation works under the hood
- How batching and streaming can coexist in the same system
- Where the key performance bottlenecks tend to appear in LLM serving

Let's dive into the service architecture to get a high-level overview of how this sample service is structured and how its core components fit together.

Service Architecture

This sample LLM serving service has six core components:

API server
: Deals with HTTP API endpoints and request/response handling

LLM engine
: High-level orchestrator, operates inference end-to-end

Workload manager
: Handles request queuing and batch management

Model executor
: Manages processes and coordinates model-worker execution

Model worker
: Executes model inference execution in its own process

Model manager
: Loads and caches the model

Figure 3-1 shows how these components come together to form the complete service. As shown, the *LLM engine* acts like an orchestra conductor—setting the stage and coordinating how everything runs. It initializes all core components, including loading the LLM model, and then orchestrates how generation requests are handled.

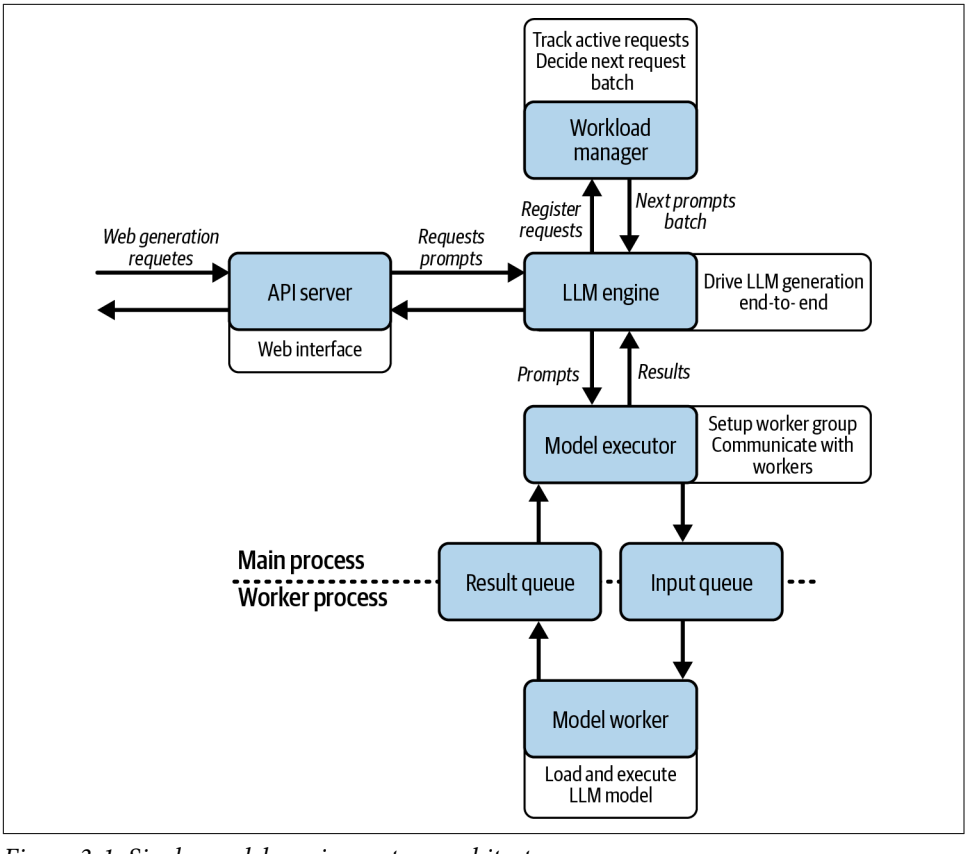

Figure 3-1. Single-model serving system architecture

Since the service is designed to manage concurrent web requests, with support for both batching and streaming, a *workload manager* is introduced to track the status of each prompt included in the generation request and determine the next batch prompts for the model to process. This is a critical point where different batching strategies can be applied to optimize the LLM throughput.

The *model executor* and *model worker* are the core components responsible for running model inference. The model worker handles model loading, hosting, and execution—running in its own separate *process*. In contrast, the model executor is responsible for initializing the model workers, setting up worker groups (including inter-worker communication), and making cross-process calls to trigger inference and retrieve generation results from the workers.

While this multi-process setup may seem like overkill for a simple example, it reflects a common and practical pattern in real-world LLM serving systems that LLMs typically run on GPU devices. It's a standard practice to isolate model execution

in dedicated processes bound to specific GPU devices. Meanwhile, the main web service remains on the CPU, focusing on handling concurrent request management and orchestration. This is because GPUs are expensive, and you don't want GPUs running idle, waiting for CPUs to perform other tasks such as input preprocessing, output postprocessing, tokenization, and de-tokenization. In a multi-process setup, all compute-intensive tasks are moved to different processes, leaving dedicated CPU resources for CPU–GPU interaction. This helps to ensure optimal GPU utilization.

With the architecture in mind, let's move on to implementing the service. We'll start by handling a single generation request, then progressively add support for batching and streaming.

Implement Single Generation Request Handling

In this section, you'll implement basic LLM model serving, handling one request (one prompt) at a time. First, initialize the core components in the `LLMEngine` class:

```
class LLMEngine:
    def __init__(self):
        self.model_executor = ModelExecutor()
        self.workload_manager = WorkloadManager()
        self.max_tokens = 20

# Initialize and set up the model worker
self.model_executor.setup_worker("facebook/opt-125m")
```

Next, set up the `ModelWorker` and `ModelExecutor` classes. In this example, you'll run a single `ModelWorker` in its own process. The model executor communicates with the worker using two event queues: `task_queue` and `result_queue`. It sends prompt requests to the worker via `task_queue` and receives generation results from `result_queue`:

```
class ModelExecutor:
    def __init__(self):
        self.task_queue = mp.Queue() # Model request queue
self.result_queue = mp.Queue() # Model result queue

    def setup_worker(self, model_name: str):
        self.worker_process = mp.Process( # Single process worker
            target=ModelWorker.run,
            args=(model_name, self.task_queue, self.result_queue)
        )
        self.worker_process.start() # Start the worker process
```

Now, let's prepare `ModelWorker`, which loads the `facebook/opt-125m` model from Hugging Face with the `AutoModelForCausalLM` class with help from the model manager:

```
class ModelWorker:
    def __init__(self, model_name: str):
        self.device = "cuda" if torch.cuda.is_available() else "cpu"
        self.model, self.tokenizer = ModelManager().load_model(model_name)
class ModelManager:
    def load_model(self, model_name: str = "facebook/opt-125m"):
        model = AutoModelForCausalLM.from_pretrained(model_name)
        tokenizer = AutoTokenizer.from_pretrained(model_name)

        return model, tokenizer
```

The worker runs a `while` loop in its own process (see the following `run` function), which continuously monitors the `task_queue`. When a request arrives in the queue, it pulls the prompt from the `task_queue`, performs model inference, and places the generation result into the `result_queue`. See the implementation as follows:

```
class ModelWorker:
    @staticmethod
    def run(model_name: str, task_queue: mp.Queue, result_queue: mp.Queue):
        worker = ModelWorker(model_name)
        while True:
            logger.debug("Waiting for batch from queue...")
            request = task_queue.get()
            result_queue.put(('complete', worker.generate(request)))
```

Now that the stage is set, let's connect all the components to implement the actual model serving logic. Refer to Figure 3-2 for an overview of the code execution workflow.

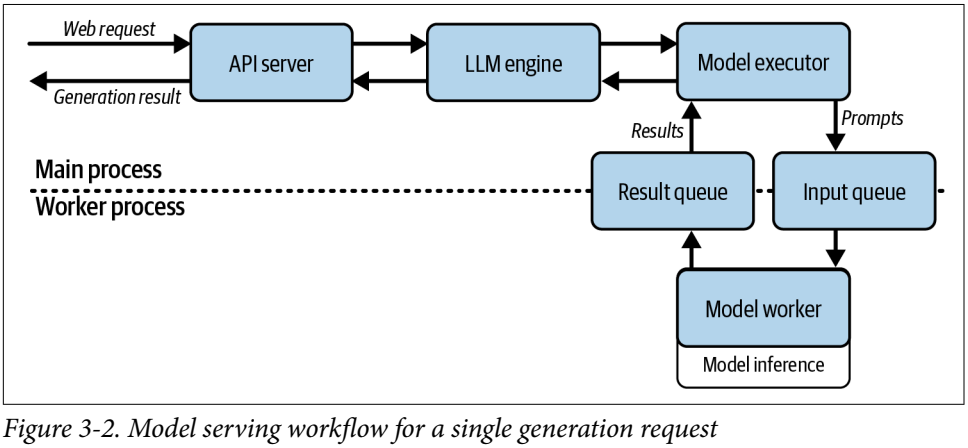

Figure 3-2. Model serving workflow for a single generation request

You'll start by defining your web API—the `basic_generate` endpoint. This endpoint accepts a single prompt as the request payload and returns the generated text as a plain string:

```
@app.post("/basic_generate", response_model=GenerateResponse)
async def basic_generate(request: GenerateRequest, \
                         llm: LLMEngine = Depends(get_llm)):
   generated_text = llm.basic_generate(request.prompt)
   return GenerateResponse(generated_text=generated_text)

class GenerateRequest(BaseModel):
   prompt: str
class GenerateResponse(BaseModel):
   generated_text: str
```

Next, you'll implement the orchestration logic in `LLMEngine`. First, the `basic_generate` function sends the prompt to the `ModelExecutor`. Then the `ModelExecutor` passes the prompt to the `ModelWorker` via the `task_queue`. When the `ModelExecutor` receives the generated result from the `result_queue`, it returns the result back to `LLMEngine`, which returns it to the web API:

```
class LLMEngine:
   def basic_generate(self, prompt: str) -> str:
       sequence = Sequence(str(uuid.uuid4()), prompt, None, None)
        # Execute the prompt generation
        results = self.model_executor.execute(sequence)
       return results[0]['generated_text']

class ModelExecutor:
   def execute_batch(self, prompt: str):
       # Send prompt to the ModelWorker's task queue
        self.task_queue.put((prompts, False))
       # Collect generation results from ModelWorker
        results = self.result_queue.get()
       return results
```

The final piece is implementing the inference logic within `ModelWorker`. In its `run` function, the `ModelWorker` picks up requests from the `task_queue`, performs model generation, and sends the results back to the `ModelExecutor` via the `result_queue`:

```
class ModelWorker:
   def generate(self, prompt: str):
       # Run model inference, model is loaded during service initialization
       outputs = self.model.generate(...)
       generated_text = self.tokenizer.decode(outputs[0], …)
       return {
           'request_id': prompt_data.id,
           'generated_text': generated_text
       }
   @staticmethod
   def run(model_name: str, task_queue: mp.Queue, result_queue: mp.Queue):
       … …
       request = task_queue.get()  # Fetch generation(prompt) task
       # Return result to ModelExecutor
        result_queue.put(('complete', worker.generate(request)))
```

With that, you've completed the first version of your online model serving service. If you compare it to the LLM model execution we explored in Chapter 2, you can see how the model serving service emphasizes building efficient software to expose model inference through a web interface.

Now you can use a few test cases to verify that the `basic_generate` API is working as expected:

```
def test_generate(client):
    response = client.post(
        "/basic_generate",
        json={"prompt": "Hello, I am"}
    )
```

If you roll out this service as is, the first challenge you'll encounter is low throughput. With the current inference service, users can send only one prompt at a time, and the model worker processes a single prompt per inference call.

As you learned in "LLM Batch Serving Basics" on page 61, the compute resources are underutilized, since LLMs are well-suited for processing multiple prompts together in a batch. In the next section, you'll upgrade the service to support batching, which will significantly improve throughput.

Batching

Now let's implement batching using a new `generate` API. This API accepts a list of prompts in a single prediction request and returns a list of generated texts in the same order (to their prompt) as the input array:

```
@app.post("/generate", response_model=BatchGenerateResponse)
async def generate(request: BatchGenerateRequest):
    generated_texts = llm.generate(request.prompts)
    return BatchGenerateResponse(generated_texts=generated_texts)

class BatchGenerateRequest(BaseModel):
    prompts: List[str]

class BatchGenerateResponse(BaseModel):
    generated_texts: List[str]
```

With the API in mind, let's discuss the general idea of batching before you implement it. Imagine you receive two requests: `request1` contains two prompts (`promptA` and `promptB`), while `request2` includes three prompts (`promptC`, `promptD`, and `promptE`). To handle these two requests in your service, you need to solve two key challenges. First, you need to combine prompts from different requests into a few batches, so the LLM can execute prompts in big batches, not per request, maximizing resource utilization. Second, you need a way to accurately map the generated outputs back

to their corresponding original requests, so the service can return the results to the correct users in the right order.

Figure 3-3 presents our updated design, extending the single-request service architecture from Figure 3-2. To support batching, in this version, we've introduced a tracking data structure called `Sequence` for each prompt and a new component called `WorkloadManager`, and we've given the LLM engine additional responsibilities.

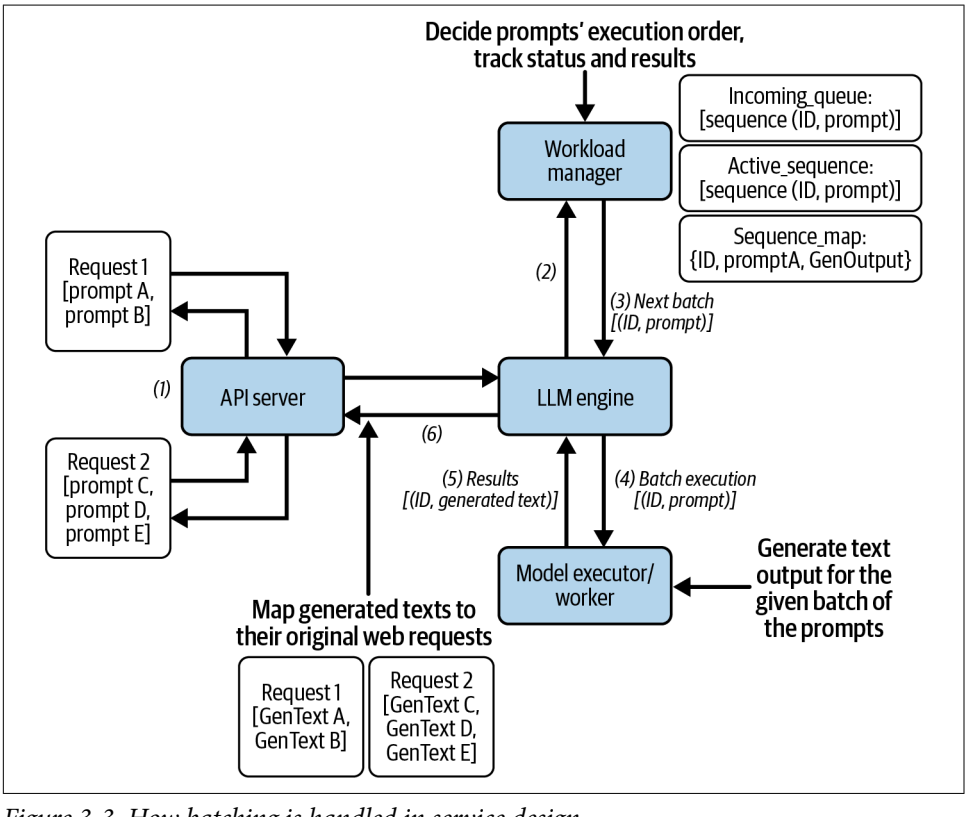

Figure 3-3. How batching is handled in service design

Follow the sequence in Figure 3-3, using the numbers in the diagram, to see how the system processes the two incoming generation requests with their multiple prompts:

1. *Request intake*
   The API server receives the requests and extracts individual prompts from each payload.

2. *Prompt queuing*
   The API server passes the requests to the LLM engine, which forwards them to the workload manager.

3. *Prompt tracking and batching*
   The workload manager assigns a unique ID to each prompt and wraps it into a `Sequence` object. The workload manager maintains in-memory tracking of all active sequences and determines which prompts should be grouped into the next batch for processing.

4. *Batch execution*
   The LLM engine retrieves the next prompt batch from the workload manager and sends it to the model executor and worker for inference.

5. *Model inference*
   The model worker processes the entire batch in one inference call and returns the generated texts, each paired with its corresponding prompt ID.

6. *Response mapping*
   The LLM engine uses the prompt IDs to map each generated text back to its originating web request. This allows it to correctly return the generated outputs—for example, mapping `GenTextA` and `GenTextB` to `request1`, and `GenTextC`, `GenTextD`, and `GenTextE` to `request2`.

With this workflow in mind, let's dive into the core implementation—starting with how each prompt is tracked and how we decide which prompts go into a batch (Example 3-1).

Example 3-1. Workload manager batch scheduling implementation

```
# Each prompt is represented as a Sequence object
class Sequence:
   def __init__(self, seq_id: str, prompt: str, client_stream, loop):
       self.id = seq_id
       self.prompt = prompt
       self.output = []
       .. .. ..

class WorkloadManager:
   self.batch_size = 4 # Set max 4 prompts in a batch

   # Create a Sequence object for each prompt and assign an ID.
   def add_request(self, prompt: str) -> str:
       request_id = str(uuid.uuid4())
       sequence = Sequence(request_id, prompt, None, None)
       self.incoming_queue.put(sequence)
       self.sequence_map[request_id] = sequence
       return request_id

     # Use first-in-first-out (FIFO) strategy
     # to generate next batch of prompts
     def get_next_batch(self) -> List[Sequence]:
```

```
        while len(self.active_sequences) < self.batch_size \
              and not self.incoming_queue.empty():
            sequence = self.incoming_queue.get()
            self.active_sequences.append(sequence)

        return self.active_sequences
```

We want to highlight two key aspects of the `get_next_batch` function. First, the workload manager uses a first-in, first-out (FIFO) strategy to determine which prompts are selected for the next batch. Second, we've capped the maximum batch size, allowing the LLM to process up to four prompts at a time.

Throughput Optimization in Batching

In practice, batch size and batching strategy have a significant impact on inference throughput. Choosing the right batch configuration requires careful tuning and is far more complex than what's shown in this simplified example. These settings must be adjusted often, based on the specific LLM model, the prompt characteristics, the nature of your web traffic, and the hardware on which the model is running.

In this example, we use a simple FIFO batching strategy to build intuition, but for a deeper exploration of batching optimization techniques such as dynamic batching and continuous batching, see Chapters 6 and 7.

Now let's look at the implementation for batch-response mapping in `LLMEngine` (Example 3-2).

Example 3-2. LLM engine batch implementation

```
class LLMEngine:
    # process multiple prompts in a request
    def generate(self, prompts: List[str]) -> List[str]:

        # Register prompt to workload manager
        # Track the prompt ID, assigned by workload manager
 prompt_ids = []
        for prompt in prompts:
            prompt_id = self.workload_manager.add_request(prompt)
            prompt_ids.append(request_id)

        # Keep executing batches until all prompts in the
        # requests are completed.
 while not self._is_batch_finished(request_ids):
            sequences = self.workload_manager.get_next_batch()
            results = self.model_executor.execute_batch(sequences)
            self.workload_manager.update(results)
```

```
        # Get generated result by using prompt_id
        generated_texts = []
        for prompt_id in prompt_ids:
            generated_texts.append(\
                self.workload_manager.get_sequence(\
                    prompt_id).output[0])
              self.workload_manager.\
                    remove_finished_sequence(request_id)

        return generated_texts
```

As shown in the `generate` function of `LLMEngine`, the `generate` logic first collects all `prompt_ids` associated with a generation request. It then uses these IDs to retrieve the corresponding generated text. The function completes only after results for all prompts are available. While batching may introduce latency for individual requests—since some may need to wait—it significantly improves overall service throughput by optimizing resource utilization across requests.

Now that you've implemented batching logic, use the following test code to verify your implementation:

```
def test_generate_batch(client):
    # Test with multiple prompts
    test_prompts = [
        "Hello, I am",
        "The weather is",
        "I want to",
        "The best way to",
        "The most efficient way to"
    ]

    response = client.post(
        "/generate",
        json={"prompts": test_prompts}
    )
```

Tracking Every Prompt's Execution

Tracking each prompt individually is a common technique in LLM serving. It decouples the user's web request from the actual model execution, allowing the system to reorganize prompts for more efficient processing in the backend. This abstraction provides the serving service with the flexibility to apply optimization techniques such as dynamic batching and prioritization (discussed in Chapter 6).

In this example, each prompt in a user request is converted into a `Sequence` object (as shown in Example 3-1), which becomes the fundamental unit for managing the LLM execution workload.

The batching implementation shown here is intentionally simplified. Its goal is to make the request-grouping logic and execution flow explicit—not to represent a fully optimized production system.

In real serving frameworks, batching is more sophisticated. Production systems use techniques such as continuous batching, dynamic scheduling, and memory-efficient KV cache management (which we will cover in Chapter 6) to maximize GPU utilization while controlling latency. For example, vLLM's continuous batching allows new requests to join an active decoding batch, improving throughput without waiting for a fixed batch to complete.

You can think of this implementation as a conceptual foundation. By understanding this baseline design, you'll better appreciate how optimized serving engines like vLLM extend and refine these ideas in practice.

Streaming with Batching

In this section, we'll add streaming support on top of the batching mechanism we just implemented. Currently, the batching code waits until all prompts are fully processed before returning results, which can introduce significant latency and lead to a poor user experience.

As we discussed in Chapter 2, LLMs generate inferences one token at a time. To reduce the latency of waiting for the entire batch to complete, we want to stream tokens back to users as soon as they are produced—while *still batching requests internally at each generation step*. This allows us to maintain high throughput while providing a more responsive real-time experience for users.

Table 3-1 outlines the new streaming and batching user experiences we plan to support.

Table 3-1. New streaming user experience

Time	Action	Backend batch	Streamed tokens
T0	User A's prompt arrives [Prompt1: "Hello, I am"]	[Prompt1]	Prompt1: "a"
T1	User B's prompt arrives [Prompt2: "I want to"]	[Prompt1, Prompt 2]	Prompt1: "a", "student" Prompt2: "see"
T2	User C's prompt arrives [Prompt3: "I like to"]	[Prompt1, Prompt 2, Prompt 3]	Prompt1: "a", "student", "[end]" Prompt2: "see", "a" Prompt3: "eat"
T3	A finishes, D joins [Prompt4: "The best way to"]	[Prompt 2, Prompt 3, Prompt 4]	Prompt2: "see", "a" Prompt3: "eat" Prompt4: "success"

In Table 3-1, you can see how the service processes prompts over time—from T0 to T3. Prompts [1–4] are received from four different users [A–D] at different times [T0–T3]. The service batches these prompts in the backend for model inference. It also tracks the generated tokens for each prompt individually. As tokens are produced, they are streamed back to users in real time. Once a prompt completes, it is removed from the backend batch to make room for new requests. For example, Prompt1 is removed at T3, since it's completed at T2.

With this streaming user experience in mind, check out Figure 3-4 to get an overview of how to implement streaming with batching in our sample service.

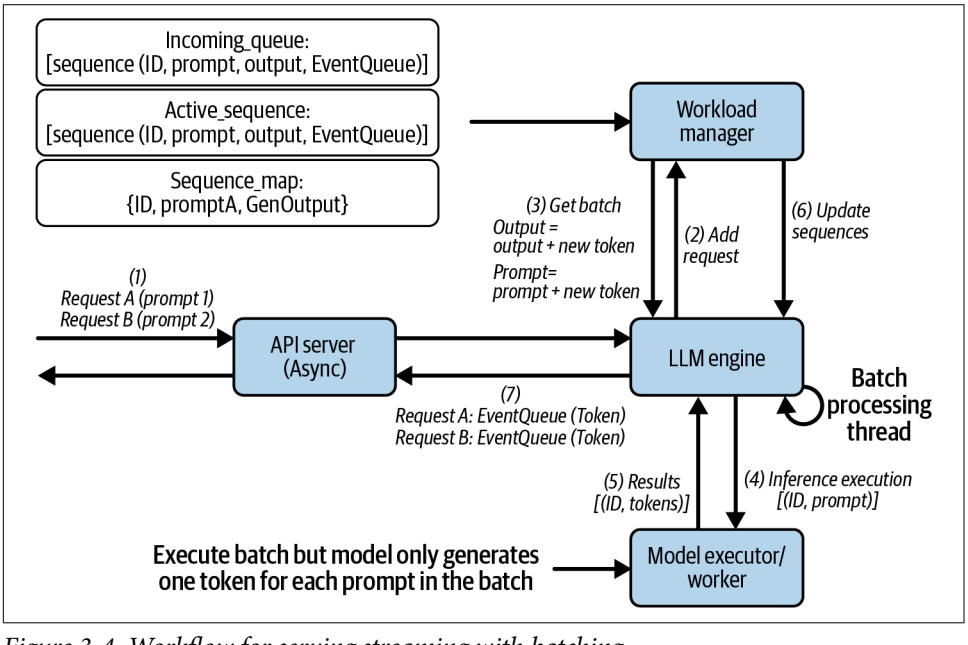

Figure 3-4. Workflow for serving streaming with batching

Let's walk through Figure 3-4 step by step:

1. *Request intake*

   The API server receives two generation requests, each containing a single prompt. Since this is an asynchronous interface, it returns a Server-Sent Events (SSE) stream, which allows the client to receive tokens incrementally as they are generated.

2. *Prompt initialization*

   The LLM engine creates a `Sequence` object for each prompt. This object has an output buffer (to store generated tokens) and an event queue (to stream tokens back to the client).

3. *Batching loop*
   The LLM engine runs a background thread that continuously orchestrates batch generation in a loop. Each loop iteration represents a single batch step for token generation.

4. *Model execution*
   A batch of prompts is sent to the model executor and worker, which runs inference and generates one *single* token per prompt in the batch.

5. *Token collection*
   The model worker returns a list of newly generated tokens, each associated with its prompt ID.

6. *Sequence update*
   The LLM engine appends each new token to its corresponding `Sequence`—updating both the output and the prompt, which will then be used for the next round of token generation.

7. *Streaming tokens*
   The LLM engine pushes the new tokens to each prompt's event queue, which signals the API server that tokens are ready to be streamed back to the client via the SSE stream.

We introduced some changes in this implementation compared to the batching implementation in Figure 3-3. The main changes we introduced are:

- The generation API is now asynchronous, allowing users to receive token updates as soon as they are generated.
- `ModelWorker` generates one token per inference step, rather than producing the entire output in a single call.
- `WorkloadManager` tracks partial outputs and updates each prompt with newly generated tokens in real time.
- Each prompt now has its own event queue, enabling `LLMEngine` to pass tokens from `ModelWorker` to the async API layer efficiently.
- `LLMEngine` includes a dedicated batch-processing thread that orchestrates token-level inference across prompts.

It's time for the core implementation. In the following code snippets, we'll focus on how generated tokens are pushed back to the client using SSE for streaming.

First, in a background thread (`requests_processing_loop`), the `LLMEngine` continuously pulls batches of prompts from the `WorkloadManager` and sends them to the `ModelExecutor` and `ModelWorker` to generate the next tokens. Here is the first part of the background thread code:

```
class LLMEngine:
   # Streaming process loop, runs in the background thread.
   def requests_processing_loop(self):
     while True:
       # Get the next batch of prompts.
active_sequences = self.workload_manager \
                                .get_next_batch(is_streaming=True)
       prompts = [{'prompt': seq.prompt, 'request_id': seq.id} \
                     for seq in active_sequences]
       # Call model executor/worker to generate tokens as a batch.
       tokens = self.model_executor.execute_forward_batch(prompts)
```

Next, in `requests_processing_loop`, once the `LLMEngine` receives tokens from the `ModelExecutor`, it routes them to the corresponding request thread in the API server using the event queue (`client_stream`) associated with each prompt. See the implementation in Example 3-3.

Example 3-3. `LLMEngine` batch-processing loop implementation

```
def requests_processing_loop(self):
        .. .. ..
    # Stream tokens back to respective clients
    for token in tokens:
       seq = self.workload_manager.get_sequence(token['request_id'])
       if result['is_finished'] or seq.token_count > self.max_tokens:
          asyncio.run_coroutine_threadsafe(
             seq.client_stream.put(None), # push finish token to request thread
             seq.loop
          )
          seq.finished = True
          self.workload_manager.remove_finished_sequence(token['request_id'])
       else:
          asyncio.run_coroutine_threadsafe(
          seq.client_stream.put(   # push token to request thread
             json.dumps({ \
                "token": token['token'], \
                "sequence_id": token['request_id']})),
             seq.loop)

          self.workload_manager.update_sequence_output( \
             token['request_id'], token['token'])
```

In each web request's handling thread, `event_generator` (see the following code), the `LLMEngine` creates an event queue for the incoming prompt and registers it with the workload manager: `workload_manager.add_streaming_request(prompt, queue, loop)`. The request can now wait for tokens on this event queue asynchronously:

```
async def event_generator(self, loop, prompt: str):
   # Create an event queue for this client's request
   queue = asyncio.Queue()
   # Register the prompt with an event queue request to workload manager
   seq_id = self.workload_manager.add_streaming_request(prompt, queue, loop)
   while True:
      # Get next token from queue
      data = await queue.get()
      if data is None:  # End of stream
         break
      yield f"data: {data}\n\n"
```

Meanwhile, the background thread, `requests_processing_loop` (see Example 3-3), continuously processes prompts in batches and dispatches generated tokens back to the corresponding event queues. This code assigns an event queue to each prompt to enable communication between individual request threads and the centralized batch-processing thread.

Finally, the API server exposes the asynchronous streaming API `generate_stream`, which waits for tokens from the `LLMEngine` and streams them to the client using SSE via a `StreamingResponse` with the `text/event-stream` content type:

```
@app.post("/generate_stream")
async def generate_stream(
    request: GenerateRequest, llm: LLMEngine = Depends(get_llm)):
   async def event_generator():
       loop = asyncio.get_event_loop()
       async for token in llm.event_generator(loop, request.prompt):
           # token = 'data: {"token": " a", "sequence_id":
           #   "8310f5e1-6f6f-480e-b2f9-c8144a12cc17"}\n\n'
          yield token

   return StreamingResponse(
       event_generator(),
       media_type="text/event-stream"
   )
```

With the streaming implementation in place, we can now test it using the following code:

```
@pytest.mark.asyncio
async def test_generate_stream(async_client):
   prompt = "Hello, I am"
   # Create a streaming request
   async with async_client.stream("POST", "/generate_stream", \
         json={"prompt": prompt}) as response:
       assert response.status_code == 200
       # Process the response stream directly
       async for chunk in response.aiter_bytes():
          # Convert bytes to string and split by newlines
          lines = chunk.decode().split('\n')
          for line in lines:
```

```
                if line.startswith("data: "):
                data = json.loads(line[6:])
                token = data["token"]
        .. .. ..
```

If the data flow in this example is still unclear, Figure 3-5 provides a visual overview of how tokens are generated and returned to the client using streaming and batching.

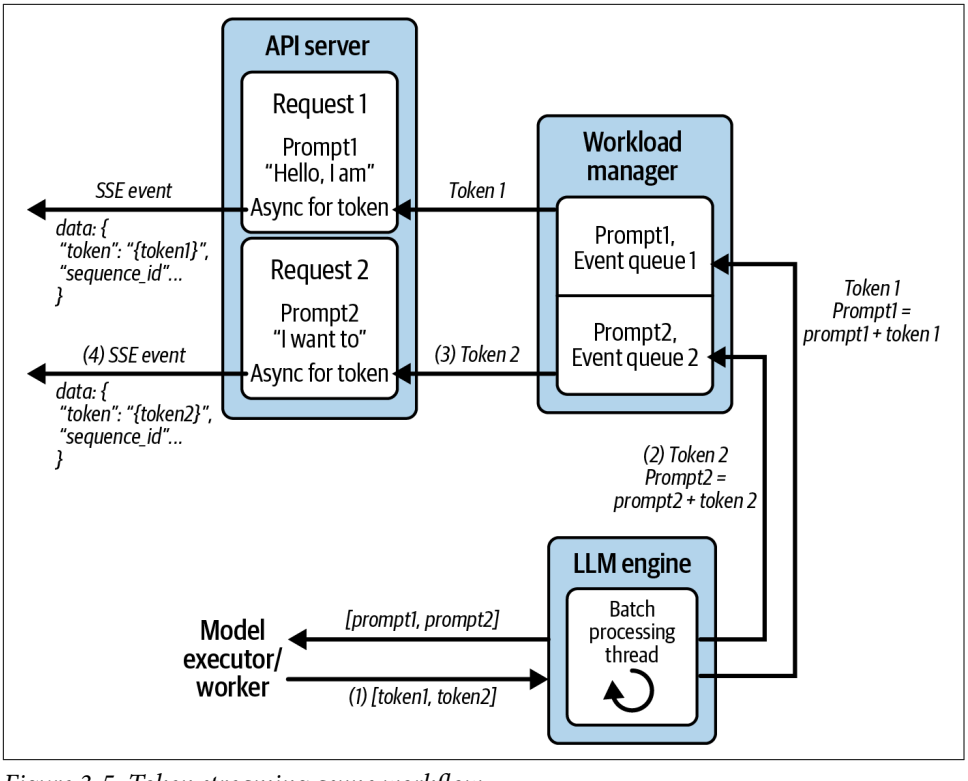

Figure 3-5. Token streaming async workflow

Figure 3-5 illustrates how the LLM processes tokens in batches while still streaming them back to individual clients in real time. The figure is designed to be self-explanatory, so you can walk through the sequence it depicts. The core idea is to track each prompt's execution using a dedicated event queue and dispatch newly generated tokens to their respective channels, ensuring timely and orderly delivery to each client.

You have now implemented all the key features of a single-model LLM serving service, including single-prompt inference, batch processing, and streaming inference. These examples should give you a sense of the substantial engineering effort involved in making LLM models accessible to end users and applications in a generic and scalable way.

In practice, building a full-fledged, production-grade model serving service—with all the complexity of thread management, event queuing, inter-process communication, and handling execution details across various model architectures—can be quite daunting. This is where model serving frameworks like vLLM come into play.

Batch Serving with vLLM

Up to this point, we have implemented batching manually to expose the internal mechanics of a serving system. Now, let's look at how a production-grade serving framework—vLLM—handles the same problem.

This section does not replace our earlier implementation. Instead, it builds on it. By first constructing a simplified batching system ourselves, we gained visibility into the coordination challenges involved in request grouping, scheduling, and concurrency control. With that foundation in place, we can now examine how vLLM abstracts and optimizes these same responsibilities.

Model serving frameworks like vLLM are designed to industrialize these mechanisms. They handle model loading, memory management, dynamic batching, streaming, and scheduling in a highly optimized manner. Rather than reimplementing these components in production, teams typically wrap or integrate such frameworks within their service architecture.

In the following example, we demonstrate how to build a single-model serving service using vLLM to provide batching and execution. The goal is to compare the two approaches: a minimal educational implementation versus a production-optimized framework. The service architecture is illustrated in Figure 3-6, and you can follow along using the code in the GitHub repository.

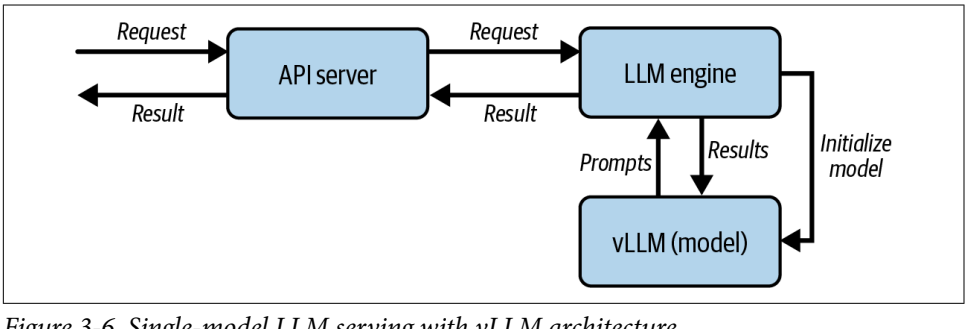

Figure 3-6. Single-model LLM serving with vLLM architecture

Compared to the streaming design shown in Figures 3-4 and 3-5, the architecture in Figure 3-6 is much more straightforward. All heavy lifting is delegated to vLLM. `LLMEngine` simply initializes vLLM with the `facebook/opt-125m` model and forwards generation requests directly to it. Look at the implementation in Example 3-4 to see how vLLM is integrated into `LLMEngine` for handling batch requests.

Example 3-4. LLM serving code with vLLM

```
class LLMEngine:
   def __init__(self):
       # Initialize vLLM with a model
       self.vllm_model = VLLM(model="facebook/opt-125m")

   def generate_vllm(self, prompts: List[str]) -> List[str]:
       # Configure sampling parameters for vLLM text generation
sampling_params = SamplingParams(
           temperature=0.7,
           top_p=0.95,
           max_tokens=self.max_tokens
       )

       # Generate text for all prompts
       outputs = self.vllm_model.generate(prompts, \
                                          sampling_params)
       # Extract generated text from outputs
       generated_texts = [output.outputs[0].text for \
                                         output in outputs]

       return generated_texts
```

In Example 3-4, you can see that using vLLM lets you enable the batch inference with just 10 lines of code. This is why such serving frameworks are so prevalent in model serving implementation.

Customizing serving frameworks helps to optimize them further. Although the vLLM framework abstracts away much of the complexity of model serving, you still need to customize it, unless you are OK with the default values. That's why it's essential to understand what happens under the hood in the serving system.

In general, knowing how model serving systems work enables you to unlock the full potential of your framework and tailor it effectively to your system's requirements, traffic patterns, and service-level objectives (SLOs/SLAs).

Also, many of the model optimization techniques introduced in Chapters 6 and 7 correspond directly to configuration options in vLLM. A solid grasp of model execution and serving fundamentals will give you the intuition you need to tune these parameters effectively. For example, when you understand how batching affects latency and throughput, you can fine-tune settings like `max_batch_size` and `max_num_seqs` to strike the right balance between real-time responsiveness and batch efficiency. Similarly, knowing how decoding leverages prompt reuse will help you optimize GPU memory allocation to support more concurrent users.

A Serving Framework Can Run as a Standalone Web Server

Many model serving frameworks, such as vLLM and SGLang, can be deployed in two different ways, depending on your use case and integration needs. You can embed your framework as a library within your custom serving application, allowing for tight integration and greater control over execution flow, or you can run it as a standalone web server, exposing a REST or streaming API that can be called by external clients or other services.

In our sample code, we adopt the library-based approach for simplicity and easier demonstration. However, real-world deployments often run the serving framework as a standalone web server mode. We will explore that pattern in more detail in the next section.

A General Design for Single-Model LLM Serving

Now that you've implemented the sample service and looked under the hood, we hope you have a solid foundational understanding of how a single LLM serving service operates. In Chapter 1, we introduced general design patterns for single-model services. However, given the importance of the concept of single-model serving—especially in the context of modern LLM serving—we'll now look closely at a design specifically tailored for LLMs.

Requirements for Single-Model Serving

The requirements any production-grade model serving system must meet generally include:

Low latency
: The system should respond quickly, completing model inference and returning results with minimal delay to maintain user engagement.

High throughput
: The system should be able to handle a high volume of concurrent requests. Maximizing throughput—measured in queries per second (QPS) or TPS—is essential for serving at scale, reducing operational costs, and improving resource utilization.

Scalability
: The system must scale horizontally to accommodate fluctuating traffic, seamlessly scaling out during peak demand and scaling in during quieter periods.

Reliability and availability
: A production model serving system must be highly available and fault-tolerant to ensure consistent and uninterrupted service.

Resource efficiency and cost management
: Serving ML models, especially LLMs on GPUs, is resource-intensive. Efficient utilization of hardware (CPU, GPU, memory) is critical to keep the cost per query manageable.

Observability
: Robust monitoring and logging are essential for tracking key performance indicators (KPIs) such as latency, throughput, and error rates, as well as model-specific metrics like prediction confidence. This visibility helps you identify bottlenecks and meet SLOs.

In addition to these general serving requirements, LLM serving introduces several unique challenges:

Large model size and memory footprint
: LLMs often require tens to hundreds of gigabytes of memory, demanding careful management of GPU resources and memory allocation.

KV cache management
: To support long context windows and efficient stateful decoding, serving systems must manage KV caches effectively across multiple requests and sessions.

Streaming responses
: For interactive applications, it's important to stream tokens back to the client as they are generated.

Concurrency and batching with variable-length workloads
: LLM requests often have varying input and output lengths. Serving systems must intelligently batch and schedule these heterogeneous workloads to maintain high throughput and GPU utilization.

LLM Serving Requirements Are Continuously Evolving

Most traditional deep-learning model serving is relatively stable and often allows the model to be treated as a black box. with standard engineering solutions for latency and scalability. In contrast, LLM serving demands more dynamic and model-aware handling.

Requirements for LLM serving frequently change as model architectures and capabilities evolve, and are often model-specific. LLM input and output lengths also vary in different user scenarios. For example, KV cache behavior varies depending on the size of the user input and the model's attention mechanism (such as multi-head versus multi-query attention).

Therefore, when designing an LLM serving system, it is crucial to isolate its evolving components from the stable infrastructure, ensuring flexibility for model-specific needs, as well as robustness in the core serving logic.

General Design

In this section, we discuss a high-level system design that addresses both the general model serving requirements and the LLM-specific challenges we have discussed. The design is intentionally kept conceptual and abstract, without relying on any specific tools, platforms, systems, or libraries. This allows it to be broadly applicable across a wide range of environments, whether deployed on private infrastructure or in the public cloud.

The core design idea is to group the aforementioned serving requirements into three distinct areas and address each separately:

Service infrastructure management
: Focuses on scaling, availability, monitoring, and efficient resource allocation

Business logic handling
: Covers customer use-case integration, request batching, and streaming responses

Model serving performance
: Targets serving latency, throughput, and optimizations for LLM-specific performance

Figure 3-7 illustrates an overall service architecture based on this separation of concerns.

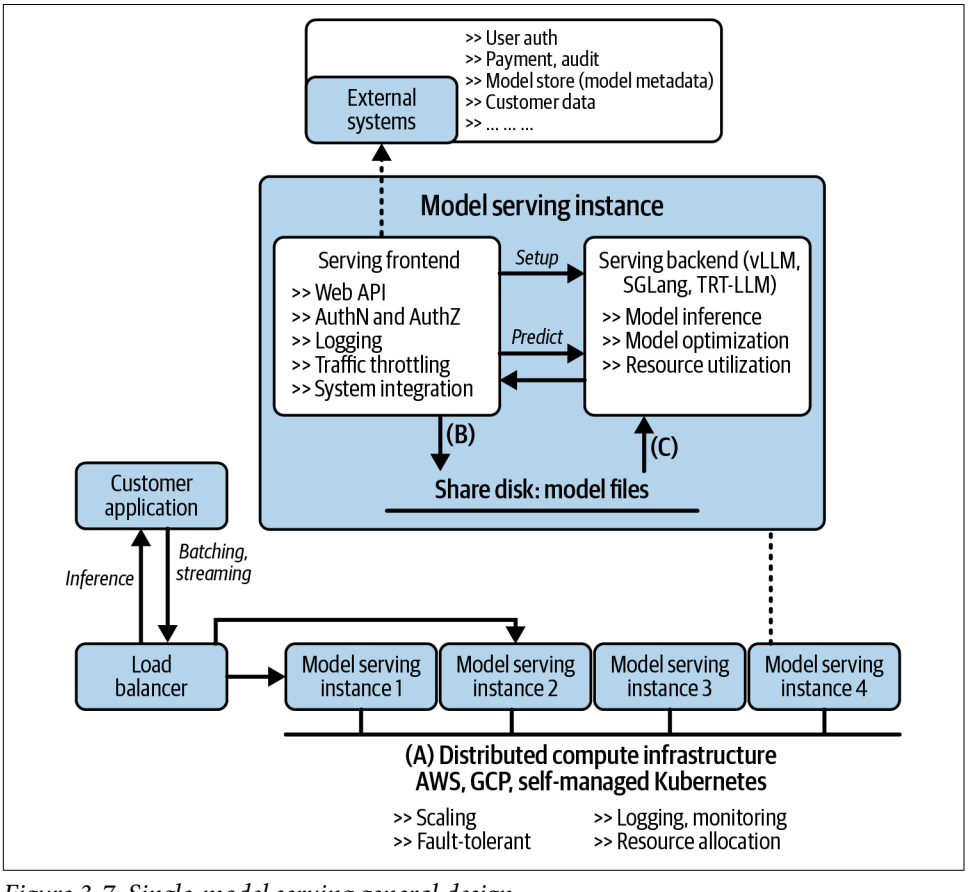

Figure 3-7. Single-model serving general design

First, in part A of Figure 3-7, we encapsulate the model serving logic into a replicable unit—such as a Docker container or a Kubernetes Pod—and delegate infrastructure-level responsibilities to a distributed compute system. This system manages key tasks such as horizontally scaling service replicas up or down based on traffic demand, restarting unhealthy instances, dynamically allocating resources, and exposing logging and monitoring interfaces.

These infrastructure-management capabilities are standard features in most modern compute platforms, including public cloud providers like AWS and Azure, as well as open source solutions like Kubernetes. By abstracting infrastructure concerns from the serving logic, we decouple model serving from low-level details such as availability, fault tolerance, and resource provisioning.

In this setup, end users do not interact directly with individual service instances. Instead, requests are routed through a load balancer, which transparently distributes traffic across available instances. This approach hides the complexity of internal scaling and enables elasticity without exposing backend details to the user.

Next, in part B, within each model serving instance is a *serving frontend* component that handles the web service interface and manages the core business logic of model serving. This layer is responsible for:

- Authenticating and authorizing requests from customer applications
- Integrating with external or internal systems, such as user customer data, model metadata, audit logs, and payment systems
- Downloading and setting up models
- Managing model configuration and runtime context
- Preprocessing inference requests, including request validation, normalization, and batching logic
- Traffic control, such as rate limiting and logging

This component acts as the middle layer between customer-facing interfaces and the backend model inference engine. This setp enables secure, flexible, and context-aware integration with business environments, for example the customer-facing chatbot, while cleanly separating customer logic from the backend inference operations handled in part C.

Part C shows the *serving backend*, the component where actual model inference takes place. This component typically runs as a separate process with access only to the serving frontend. It is entirely focused on delivering high-performance model execution.

The serving backend is designed to understand and support a wide variety of model architectures and optimization strategies. In practice, developers often rely on mature, production-grade model serving frameworks such as vLLM and Triton, which offer:

- High throughput and low latency inference engines tailored for LLMs
- Advanced optimization techniques, including quantization, KV cache management, and continuous batching of variable-length sequences
- Efficient GPU utilization, maximizing hardware performance at scale
- Ongoing evolution and strong community support, ensuring alignment with the latest model advancements

Isolating model execution in this backend component enables a modular, scalable architecture that cleanly separates business logic from inference performance concerns.

In the next chapter, we'll explore some real-world examples that demonstrate how this single-model serving paradigm works in practice. The next part of this chapter focuses on a sample multi-model service.

Build a Multi-Model Serving Service from Scratch

So far, we've focused on single-model serving. This design works well when a service hosts one model with predictable traffic and dedicated hardware. However, in many real-world systems, this assumption no longer holds.

As applications grow, you may need to serve multiple models simultaneously—different model sizes, versions, or task-specific models (such as for classification, embedding, and generation). Deploying a separate service for each model can quickly lead to resource underutilization, higher infrastructure costs, and increased operational complexity. GPUs may sit idle when certain models receive low traffic, while others become bottlenecks.

This is where multi-model serving becomes necessary rather than optional.

In this hands-on section, you will learn the essential concepts behind building a multi-model serving service. Unlike a single-model service, which hosts only one model instance, a multi-model service can dynamically manage and route requests to multiple models within a shared infrastructure. This approach is particularly effective when serving many smaller or fine-tuned models, enabling better hardware utilization and lower operational overhead.

We'll begin by implementing a simplified multi-model service with custom backends to illustrate the core principles: model loading, routing, isolation, and resource sharing. Then we'll transition to using a production-grade backend—NVIDIA Triton Inference Server—to demonstrate how these same concepts are applied in real-world systems.

After this implementation, we will examine two general multi-model serving design patterns: one optimized for cost efficiency and another optimized for latency and scalability. These variations highlight the trade-offs that shape real production architectures.

Design Goals

In this design, we'll build a model serving service that hosts three models: two Transformer-based language models and one image-classification model. All three will run on CPU. The goal is to use just enough core components to help you grasp several critical aspects of multi-model serving, including:

Cross-framework support
: How models built with different machine learning frameworks (such as PyTorch and ONNX) and architectures (such as NLP and Vision) can be hosted under a unified system.

Unified API interface
: How a single service can expose a consistent web interface (for example, a REST API) for different types of models—regardless of whether the underlying task is text generation or image classification.

Resource management
: How to manage and allocate limited compute resources (CPU, memory, and optionally GPU) when hosting multiple models on a single server instance, including strategies like lazy loading and least recently used (LRU)-based model eviction.

Let's start with a high-level overview of how the system is structured and how its core components interact.

Service Architecture

This design consists of the following five core components:

API server
: Exposes HTTP endpoints and handles request/response processing

Model manager
: Manages the model cache and coordinates model worker lifecycles

Model store
: Stores and retrieves model metadata

Model engine
: Responsible for creating model worker instances based on metadata provided by the model store

Model worker
: Loads the model into memory and handles inference execution

Figure 3-8 illustrates how these components work together to form the complete multi-model serving service.

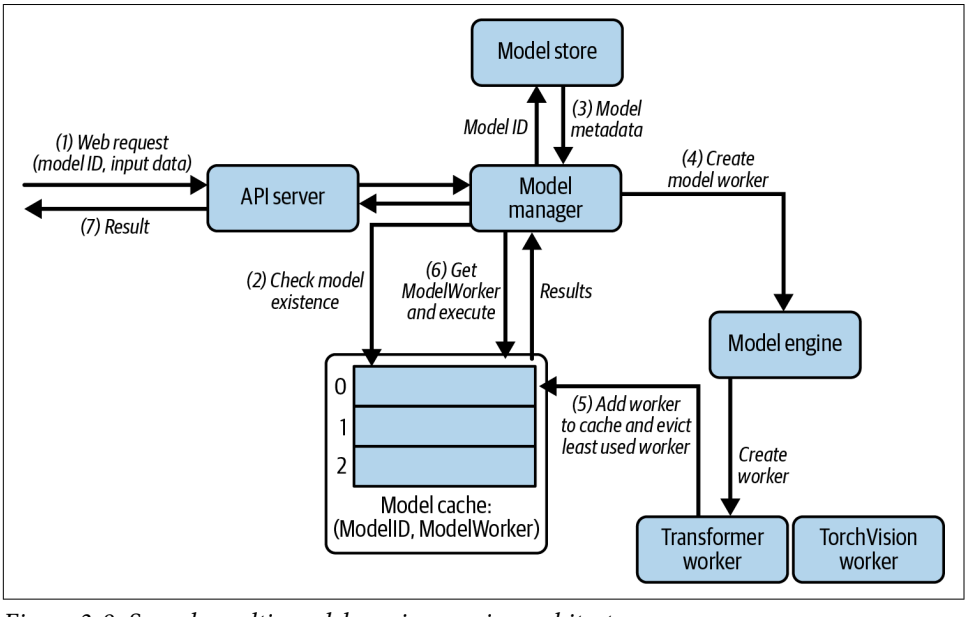

Figure 3-8. Sample multi-model serving service architecture

Now let's walk through the typical serving workflow illustrated in Figure 3-8:

1. Client request

The client sends a prediction request to the API server specifying a model ID and input payload. For example:

```
model_id = "550e8400-e29b-41d4-a716-446655440000"
input_data = "This movie was great! I really enjoyed it."
```

2. Model lookup

The API server's request handler forwards the request to the `ModelManager`, which checks whether the model is already loaded in the model cache.

3. Metadata fetch (if not cached)

If the model is not found in cache, `ModelManager` queries the `ModelStore` for the model's metadata, which defines how to load and host the model. A sample metadata entry might look like:

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "distilbert-base-uncased-finetuned-sst-2-english",
  "type": "text",
  "framework": "transformers",
  "version": "1.0.0",
```

```
        "description": "Sentiment analysis model"
    }
```

4. *Model worker creation*
   `ModelManager` passes the metadata to the `ModelEngine`, which creates a `ModelWorker` responsible for loading the model into memory and preparing it for inference.

5. *Worker registration*
   Once the `ModelWorker` is initialized, the `ModelManager` registers it in the model cache. If the cache is full, the `ModelManager` will evict the least recently used `ModelWorker` to make room for the new one.

6. *Inference execution*
   The API server retrieves the appropriate `ModelWorker` from the `ModelManager` and calls it to perform inference using the client's input data.

7. *Response*
   The API server returns the inference result to the client.

As you can see, the multi-model serving workflow is conceptually straightforward: receive a model request, load the requested model if needed, and execute inference. The core engineering effort lies in building a unified interface to host and operate models trained in different architectures and frameworks, while effectively managing the model cache to optimize compute resource usage.

Core Implementation

Let's look at some key implementations, starting with prediction request handling from the API server:

```
class PredictionRequest(BaseModel):
    model_config = ConfigDict(protected_namespaces=())
    model_id: str
    input_data: Any

@app.post("/predict")
async def predict(request: PredictionRequest):
    # Get model worker
    worker = model_manager.get_model_worker(request.model_id)
    # Make prediction
    try:
        result = worker.predict(request.input_data)
        return result
```

In the `predict` API implementation, the `ModelWorker` is retrieved from the `ModelManager` and then used to execute the model inference. Let's take a closer look at how the `ModelManager` handles model retrieval:

```
class ModelManager:
    def __init__(self, model_store: ModelStore, max_models: int = 2):
        self.model_store = model_store
        self.max_models = max_models
        self.model_cache = OrderedDict()  # model id -> worker
         self.model_engine = ModelEngine()

    def get_model_worker(self, model_id: str) -> Optional[ModelWorker]:
        # Check if model is in cache
        if model_id in self.model_cache:
            # Move to end (most recently used)
            self.model_cache.move_to_end(model_id)
            return self.model_engine.get_worker(model_id)

        # Get model metadata
        model_metadata = self.model_store.get_model(model_id)
        if not model_metadata:
            return None

        # Check if we need to remove least used model
        if len(self.model_cache) >= self.max_models:
            # Remove least recently used model
            id, model_worker = self.model_cache.popitem(last=False)
            self.model_engine.delete_worker(id)
        # Create and cache new model worker
        self.model_cache[model_id] = \
                  self.model_engine.create_worker(model_metadata)
        return self.model_cache[model_id]
```

In the `get_model_worker` function, you can see that the primary role of the `Model Manager` is to maintain a model cache—controlling which models are loaded into memory and how many can be kept active at once. In this example, we use a simple LRU strategy: when the number of cached models exceeds the `max_model_count`, the least recently used model is evicted to free up memory.

Next, let's take a look at how the `ModelWorker` loads and initializes models:

```
class ModelEngine:

    def create_worker(self, model_metadata: ModelMetadata) -> ModelWorker:
        if model_metadata.id not in self.workers:
            if model_metadata.framework == "transformers":
                self.workers[model_metadata.id] = \
                          TransformerWorker(model_metadata)
            elif model_metadata.framework == "torchvision":
                self.workers[model_metadata.id] = \
                          TorchVisionWorker(model_metadata)
            elif .. ..
        return self.workers[model_metadata.id]
```

In the `create_worker` function, the `ModelEngine` selects the appropriate type of `ModelWorker` based on the model's framework (for example, `TorchVision`). This example implements two types of workers—`TransformerWorker` and `TorchVision Worker`—to demonstrate how different model backends are required to support a variety of model types. Let's take a closer look at the implementation of the `_load_model` and `predict` functions of `TransformerWorker` in the following code:

```
class TransformerWorker(ModelWorker):
    def __init__(self, model_metadata):
        self.tokenizer: Optional[AutoTokenizer] = None
        super().__init__(model_metadata)

    def _load_model(self):
        if self.model is None:  # Only load if not already loaded
            self.model = AutoModelForSequenceClassification.\
                         from_pretrained(self.model_metadata.name)
            self.tokenizer = \
                AutoTokenizer.from_pretrained(self.model_metadata.name)

    def predict(self, input_data: Any) -> Dict[str, Any]:
        # Tokenize the input
         inputs = self.tokenizer(input_data, return_tensors="pt", \
                                 padding=True, truncation=True)
        with torch.no_grad(): # Run model inference
            outputs = self.model(**inputs)
        predictions = torch.softmax(outputs.logits, dim=-1)
        return {"predictions": predictions.tolist()}
```

Now, let's look at the model metadata logic. From the previous `ModelManager` and `ModelEngine` code, you can see that the system relies on each model's metadata to determine the appropriate `ModelWorker` for handling an inference request. In our simplified example, the model metadata is defined as follows:

```
class ModelMetadata(BaseModel):
    id: str
    name: str
    type: str
    framework: str
    version: str
    description: str

class ModelStore:
    def __init__(self, config_path: str):
        self.models: Dict[str, ModelMetadata] = {}
        self._load_config(config_path)

    def _load_config(self, config_path: str):
        with open(config_path, 'r') as f:
            config = json.load(f)
            for model in config['models']:
                self.models[model['id']] = ModelMetadata(**model)
```

In real-world applications, model metadata is typically stored in a remote database or metadata service designed specifically for managing models. However, here we use the `ModelStore` class to host metadata, which we load from a local JSON file for simplicity. Here's a sample from that JSON file:

```
"models": [
       {
           "id": "550e8400-e29b-41d4-a716-446655440000",
           "name": "distilbert-base-uncased-finetuned-sst-2-english",
           "type": "text",
           "framework": "transformers",
           "version": "1.0.0",
           "description": "Sentiment analysis model"
       },
       {
           "id": "7c9e6679-7425-40de-944b-e07fc1f90ae7",
           "name": "pytorch/vision:mobilenet_v2",
           "type": "image",
           "framework": "torchvision",
           "version": "1.0.0",
           "description": "Image classification model"
       },
       .. .. ..
```

Let's recap how our sample service addresses the three key design requirements:

- To support various types of models, we implement different `ModelWorker` classes, each designed to handle the loading and execution logic for a specific model type.
- To accommodate different model input formats, the `predict` interface uses a generic input and output structure, which is agnostic to the model used in the prediction request. We assume the client is responsible for preparing (preprocessing) the input and interpreting (postprocessing) the output, as they are expected to understand the model they are invoking.
- For resource management, models are loaded into memory on demand upon receiving requests. We use an LRU cache to manage memory consumption—evicting the least-used models when memory usage exceeds a predefined threshold.

In practice, maintaining backend support for loading and executing a wide variety of models, managing model metadata and configurations, and coordinating thread safety and concurrency can be complex and error-prone. For these reasons, it's often more efficient to delegate these responsibilities to a dedicated multi-model serving framework. This allows you to focus your own efforts on integrating that framework into your business applications.

Next, let's explore how to leverage the NVIDIA Triton Inference Server to replace the `ModelEngine` and `ModelWorker` components in our current example.

Using NVIDIA Triton as a Model Server

In "Multi-Model Service" on page 24, we introduced the general concept of multi-model serving and highlighted NVIDIA Triton as one of the most popular and widely adopted multi-model solutions. It offers a standardized, high-performance way to serve models built with different formats—such as PyTorch, TensorFlow, ONNX, and TensorRT—through a consistent HTTP/gRPC API.

In this section, we'll extend our multi-model serving example to demonstrate how to integrate Triton as the model serving backend. But let's first take a quick look at how Triton works in Figure 3-9.

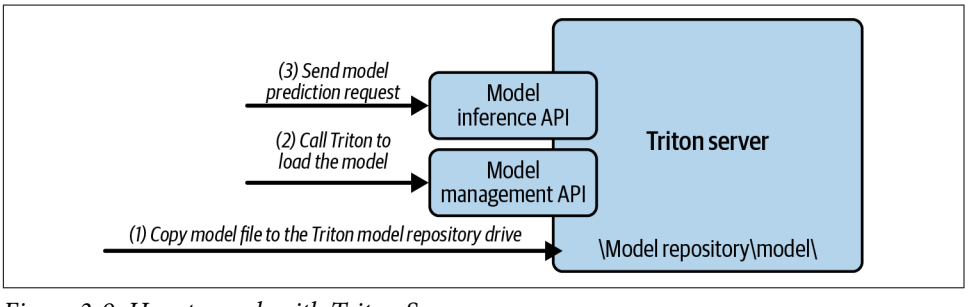

Figure 3-9. How to work with Triton Server

As Figure 3-9 shows, the Triton Server runs as a web service and exposes two key types of APIs: a model management API for loading, unloading, and configuring models, and a model inference API for sending prediction requests.

It takes just a few straightforward steps to run model inference with Triton:

1. Copy the model (in a Triton-supported format) into the designated model repository directory on the server, such as */models/densenet_onnx/*.
2. Load the model using Triton's management API. For example:

   ```
   curl -X POST http://localhost:8000/v2/repository/models/densenet_onnx/load
   ```

3. This command instructs Triton to load the densenet_onnx model from the repository if it exists and is properly configured.
4. Send an inference request using the Inference API. For example:

   ```
   curl -X POST http://localhost:8000/v2/models/densenet_onnx/infer
   ```

5. This triggers prediction on the loaded model using the input data provided in the request payload.

Now, let's see how to integrate Triton Server into the multi-model service. Compared to the previous multi-model design (shown in Figure 3-8), the updated architecture in Figure 3-10 introduces two new components: `TritonWorker` and `TritonServer`.

The Triton worker acts as a wrapper that handles model loading and inference requests through the Triton Inference Server, which runs as a separate web service in its own process.

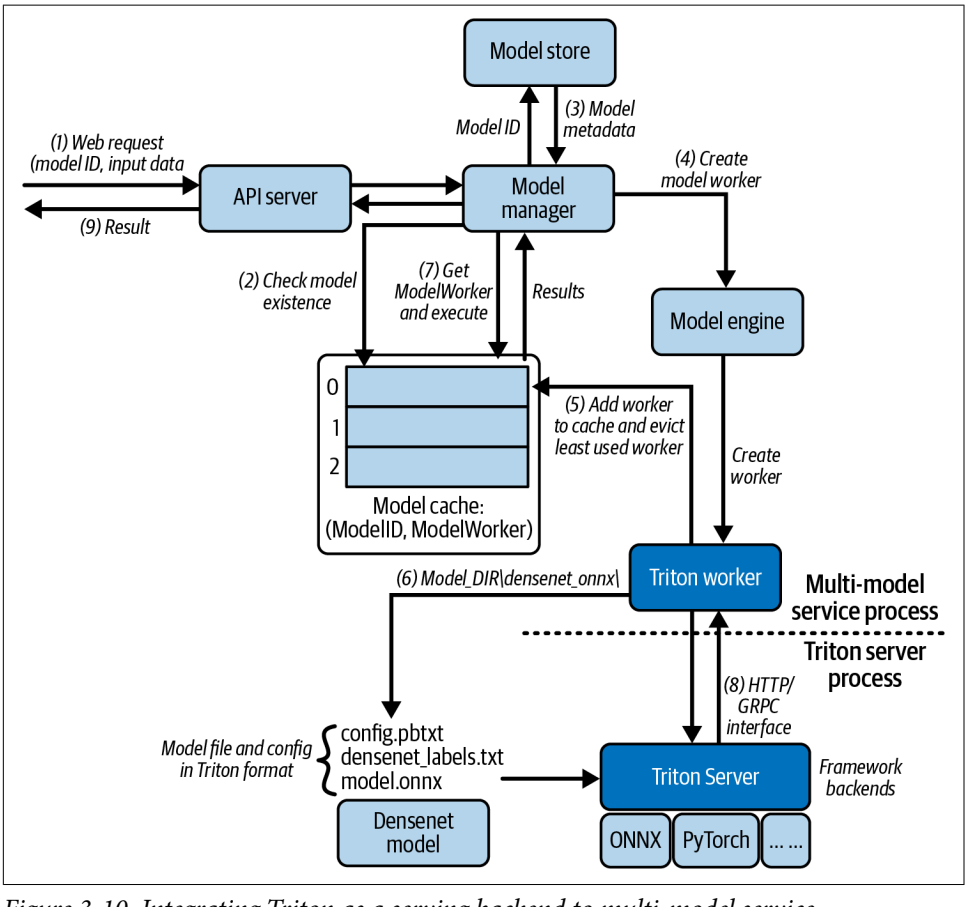

Figure 3-10. Integrating Triton as a serving backend to multi-model service

In this design, we delegate model hosting and execution responsibilities completely to Triton, while retaining model cache management, model file handling, and the external web interface within the multi-model service layer.

Although simplified, this design illustrates a common pattern in practice: using a wrapper service for handling the client web requests, integrating dependent systems, and managing resources (such as memory and CPU/GPU) and the model file lifecycle, while offloading the core inference workload to Triton.

Now let's look at the code implementation for `TritonWorker`, starting with the model initialization logic. The `TritonWorker` will load the given model into Triton through the Triton management API, using the `_load_model` function as follows:

```
class TritonWorker(ModelWorker):
   def __init__(self, model_metadata):
      self.triton_url = "0.0.0.0:8009"  # Triton server URL
         self.client = httpclient.InferenceServerClient(\
                                    url=self.triton_url)

   def _load_model(self):
       """Load model to Triton through Triton management API"""
  load_url = f"http://{self.triton_url} \
          /v2/repository/models/{self.model_metadata.name}/load"

       response = requests.post(load_url)
```

Then let's look at the inference code. In the following `predict` function, the logic first converts the input data to the Triton format (which is declared in the model's configuration), then sends a prediction request to Triton Server (`self.client.infer()`) to run model inference:

```
def predict(self, input_data: Dict[str, Any]) -> Dict[str, Any]:
   """Make prediction through Triton inference API
   Args:
     input_data: Dictionary containing input tensors for
                 the model
     Each key should be an input name and value should be a
        numpy array Example: {"data_0": np.array(...)}

     Returns:
       Dictionary containing output tensors from the model
         Each key is an output name and value is a numpy array
       """
    # Create input tensors
    inputs = []
    for name, data in input_data.items():
       if not isinstance(data, np.ndarray):
       # Convert list or other array-like data to numpy array
         shape = data["shape"]
         content = data["data"]
         # Explicitly set dtype to float32
         array = np.array(content, \
             dtype=np.float32).reshape(shape)
       else:
         # Ensure existing numpy array is float32
             array = data.astype(np.float32)
         input_tensor = httpclient.InferInput(name, \
                                    array.shape, "FP32")
         input_tensor.set_data_from_numpy(array)
         inputs.append(input_tensor)
```

```
        # Hardcode output name for DenseNet model
        output_name = "fc6_1"

        # Make inference request
           response = self.client.infer(
           model_name=self.model_metadata.name,
           inputs=inputs,
           outputs=[httpclient.InferRequestedOutput(output_name)]
        )

        # Get predictions and convert numpy arrays
        # to lists for JSON serialization
        predictions = {
           output_name: response.as_numpy(output_name).tolist()
        }
       return predictions
```

Finally, let's clean up resources on the Triton Server by sending a model unload request. Resource cleanup is a critical part of any multi-model serving system, as it allows you to reclaim limited compute and memory resources and make them available for serving other models:

```
    def __del__(self):
        """Cleanup: unload model when worker is destroyed"""
        try:
          unload_url = f"http://{self.triton_url} \
              /v2/repository/models/{self.model_metadata.name}/unload"
          requests.post(unload_url)
```

Now that we've walked through the self-developed multi-model serving system, you can refer to our GitHub repository for detailed service setup instructions and local execution steps.

Trade-offs in Multi-Model Serving Designs

This section dives deeper into two key approaches to multi-model service design—cost-optimized and latency-optimized.

Challenges

You've seen that multi-model services are designed to co-host multiple models within a single serving instance or container, allowing GPU, CPU, and memory resources to be shared efficiently. They support dynamic model loading and unloading based on real-time traffic, enabling significant cost savings and improved price-performance.

Multi-model serving is particularly useful when you've produced many models but don't need to use them all simultaneously. For instance, let's say you have scheduled processing jobs that only run for three or four hours a day. Instead of dedicating a separate container to each model, you can load the model on demand, run the job, and then unload it, freeing up resources to serve other models or tasks. As another example, perhaps you need to train 1,000 models for 1,000 customers. Since not all models will be used at the same time, you can limit the system to host at most 200 models concurrently, then load models on demand as requests arrive.

While the cost-saving benefits of multi-model serving are clear, its two biggest real-world challenges both lie in user experience:

Cold start latency
: When a request arrives for a model that isn't currently loaded, the serving instance must download the model file, load it into memory, and possibly evict another model if the cache is full. This process can introduce several seconds—or even tens of seconds—of delay, degrading the user experience. In high-traffic scenarios, where many models are cold, this can lead to request timeouts and cascading failures in downstream applications.

Hot model scaling
: When a particular model suddenly receives high traffic, its latency increases as the load grows. Scaling that model becomes nontrivial—replicating the model across instances and updating the routing layer is complicated by the fact that each instance has an independent model cache. This creates engineering complexity and introduces nondeterministic behavior in performance.

There is no one-size-fits-all solution. As engineers, we constantly make trade-offs. The two multi-model serving approaches we examine next are each optimized for a different priority: cost and latency.

A Cost-Optimized Multi-Model Design

Let's first look at a cost-efficient multi-model architecture (Figure 3-11) that derives from the paradigm in Chapter 1 (see Figures 1-8 and 1-10), incorporating additional details and refinements based on the concepts explored in this chapter.

The design in Figure 3-11 optimizes for cost efficiency, with a focus on sharing serving resources across a variety of models, while attempting to mitigate cold start latency and hot model scaling issues.

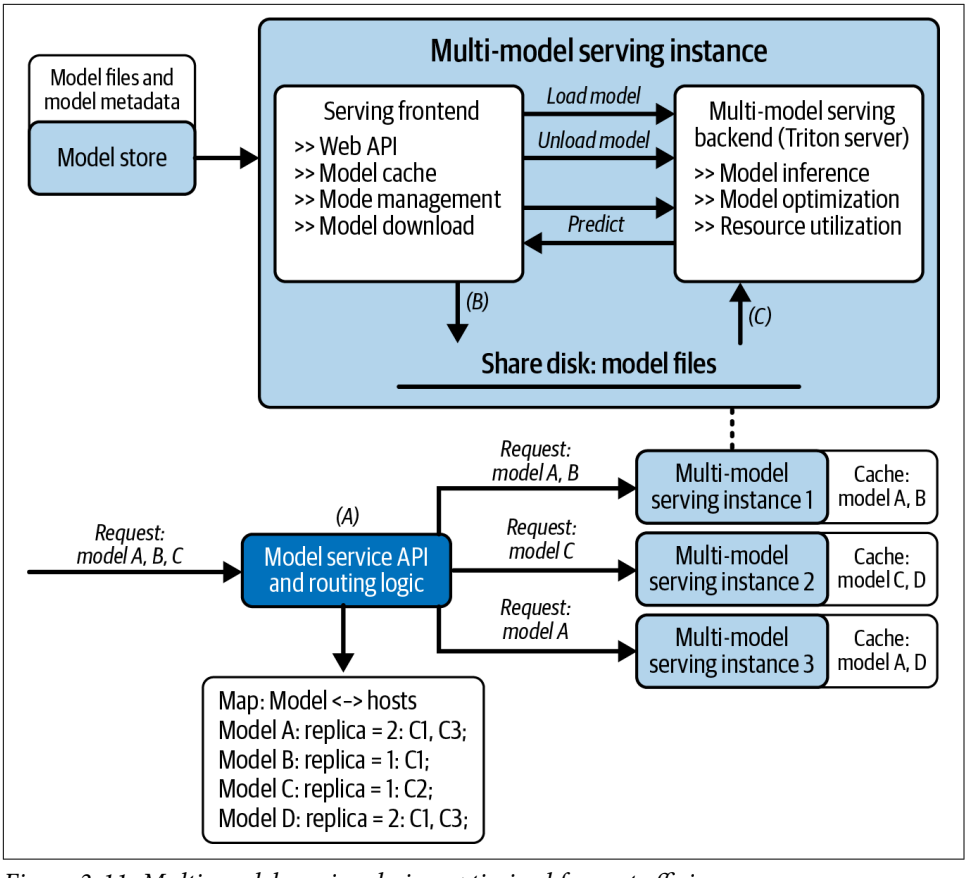

Figure 3-11. Multi-model serving design optimized for cost efficiency

In this architecture, all logic related to model loading, memory management, and hosting is encapsulated within each multi-model serving instance (parts B and C of Figure 3-11). The serving frontend handles web APIs, caching, and model management, while the backend is responsible for inference execution and resource optimization. In practice, most developers rely on model server solutions such as Triton for the serving backend.

A key part of this design is shown in part A: the model service API and routing logic. This component maintains mappings between models and serving instances, allowing it to:

- Route requests to instances that already have the model loaded, minimizing cold starts.
- Track the number of replicas per model and distribute models across instances, to scale hot models horizontally.

- Distribute model traffic efficiently on the backend by applying bin-packing strategies to load models onto the minimum number of servers, optimizing resource usage.

However, this approach is not without challenges. The system is reactive—it adjusts to traffic patterns after they emerge. This means it's always playing catch-up with demand, and prediction latency increases unavoidably when there are sudden traffic surges. Managing the routing logic, scaling decisions, and cache state across instances introduces complexity, making operations, debugging, and maintenance more difficult.

A Latency-Optimized Multi-Model Design

Let's look at a multi-model approach that favors latency. One common strategy in distributed systems engineering is to trade off capacity for improved performance. In this design, rather than sharing resources across multiple models, we provision a dedicated resource group for each model. This resource-provision approach enables better scalability and significantly reduces latency in the cold-start scenario, which works very well for high-demand models. See Figure 3-12 for the service architecture.

The primary difference between the designs in Figures 3-11 and 3-12 is that we replace the multi-model serving instances with dedicated single-model serving instance groups—one instance group per model (part A).

Another key change is that models are no longer loaded on demand. Instead, they are pre-provisioned by a model-provisioning service (part C). Before sending prediction requests, the client must first call this provisioning service to create the instance group for the target model. Once provisioned, the service updates the model-to-instance-group mapping in the routing map. When a client sends a prediction request, the model service API consults the routing map and forwards the request to the appropriate pre-provisioned instance group for that model.

This design excels in latency and scalability, especially for hot models, since each model has its own dedicated, always-on resource pool. There's no cold-start delay, and models can scale independently. Operators can also define separate resource policies for different models, providing greater flexibility in resource management and operations. The architecture is also simpler, making it easier to maintain and troubleshoot than the more dynamic, cache-heavy approach in the cost-optimized version.

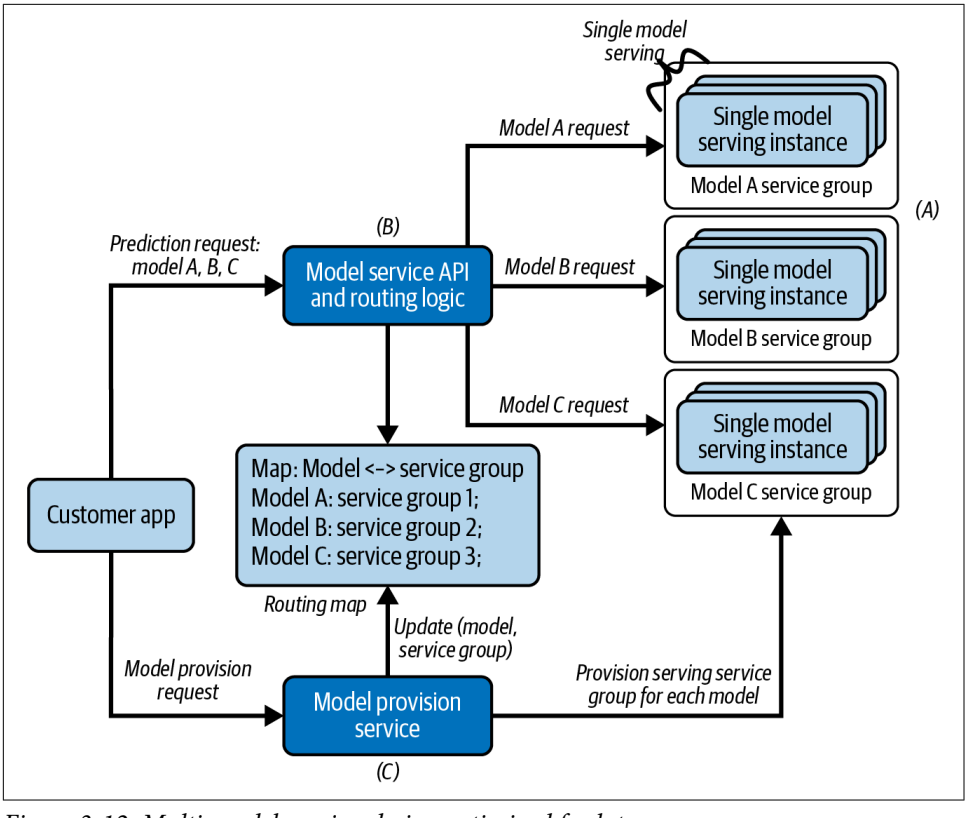

Figure 3-12. Multi-model serving design optimized for latency

The main trade-off is cost efficiency. In many cases, it's difficult to predict which models will receive heavy traffic. As a result, you may end up overprovisioning resources for underutilized models, wasting compute capacity. Still, this approach is highly effective when you're serving multiple models with steady and predictable demand.

Ultimately, we hope this comparison conveys that once you understand the fundamentals of model serving, you can tailor your architecture to meet your specific goals—whether that's cost, performance, or operational simplicity.

Multi-Model Serving Applies to LLMs, Too

While LLM serving is typically implemented using a single-model approach—primarily due to its substantial compute and memory requirements—the multi-model serving paradigm remains highly relevant for many LLM use cases. In cases such as prefix caching and routing and multiple Low-Rank Adaptation (LoRA) support, multi-model serving provides significant advantages.

For prefix caching and routing, requests with shared prompt prefixes can be routed to specific model replicas that have the corresponding KV cache already populated, reducing redundant computation and improving both throughput and latency.

Multi-model serving enables dynamic management and loading of multiple LoRA adapters on top of a shared base model, allowing for scalable, memory-efficient personalization across tenants or use cases.

We will discuss prefix caching in Chapter 7 and multi-LoRA in Chapter 10.

Summary

In this chapter, we walked you through the code and designs to build two model serving services from the ground up—introducing both single-model and multi-model serving architectures. Our goal was to help you develop practical intuition and hands-on skills for designing and implementing systems that expose ML models, especially LLMs, via web-based APIs.

We began by reviewing a simplified, self-developed single-model serving system that supports both batch processing and streaming. This walkthrough covered key components such as the API server, workload manager, model executor, and model worker—highlighting how they work together to manage concurrent requests, execute inference, and return generated outputs in real time. We also introduced advanced techniques like prompt tracking, token streaming, and concurrent batching.

To contrast and complement our custom implementation, we demonstrated how to leverage vLLM, a production-grade model serving framework, to abstract away much of the complexity while still allowing fine-grained performance tuning. This comparison underscored the value of understanding what happens under the hood, so you can tune framework settings effectively and make the right architectural decisions for your use case.

Next, we transitioned to multi-model serving, showing how a service can dynamically manage and serve multiple models in a shared infrastructure. You built a custom system that supports on-demand model loading, model metadata management, and memory-efficient model caching using an LRU strategy. This architecture supports cross-framework models like NLP and Vision, all under a unified API interface.

To further reduce complexity and improve scalability, we demonstrated how to integrate NVIDIA Triton Inference Server as the backend for multi-model inference, offloading core responsibilities like model hosting, execution, and optimization.

Finally, we explored two key multi-model serving designs: one optimized for cost efficiency through shared resources and dynamic model loading, the other optimized

for latency and scalability. These design patterns illustrate common trade-offs in model serving, balancing cost, performance, scalability, and operational complexity.

We hope you now feel more confident about both building and evaluating model serving solutions. With this foundational understanding, you're now well-equipped to tailor serving architectures to meet your application's unique requirements.

In the next chapter, we'll apply what you've learned to some real-world case studies—examining how to set up model serving infrastructure with both public cloud and open source offerings.

CHAPTER 4
Model Serving Best Practices

In Chapters 2 and 3, we explored model inference, from implementation to system design. You saw how LLMs execute internally and how to build a serving service from first principles. This chapter shifts the focus from how to build a serving system to how serving systems must evolve in real-world LLM applications.

Modern LLM applications rarely consist of a single request–response model invocation. Instead, models are embedded inside agentic workflows, enterprise platforms, and layered production systems. When this happens, model serving stops being just an inference problem—it becomes a system architecture problem. This chapter examines what changes at that system level.

We begin with agentic applications—not because this is an "agent chapter," but because agents are now the primary pattern for building LLM-powered systems. Most modern LLM use cases—knowledge assistants, copilots, workflow automation, reasoning engines—follow an agent-like structure. A single user interaction may trigger multiple LLM calls, retrieval steps, tool execution, and iterative reasoning. These behaviors fundamentally reshape serving requirements.

Agents increase token usage, amplify tail latency across chained calls, introduce dynamic compute patterns, and require orchestration across models and tools. Many of the serving optimizations discussed later in this book—caching strategies, batching approaches, memory management, scheduling, and parallelism—are motivated directly by these agentic workloads.

Understanding agents, therefore, provides the right context for both the architectural best practices in this chapter and the optimization techniques in the chapters that follow. By unpacking the core components of a working knowledge agent, we show how LLM serving powers intelligent planning, tool use, and multistep reasoning—and what that means for system design.

We then zoom out to present a layered reference architecture for an enterprise LLM serving platform. This framework highlights the essential building blocks and operational considerations that separate a hobby stack from a production-grade system. Here, serving becomes more than model execution: it includes orchestration, resource management, observability, governance, and cost control.

Next, we examine build-versus-cloud choices as a spectrum, not a binary switch. The goal is not to advocate one approach over another, but to equip you with the mental model to make situational decisions. You'll see how teams combine open source stacks with managed services, why the "right" balance shifts as traffic and cost profiles evolve, and how understanding how to build empowers you to decide how much to customize.

Finally, we close with the core performance metrics—latency, throughput, and related measures—that provide the lens through which all architectural decisions in this chapter should be evaluated. These metrics lay the groundwork for the optimization chapters that follow. By the end of this chapter, you will:

- Recognize how agentic workflows reshape serving requirements
- Understand an enterprise-ready layered serving architecture and its trade-offs
- Navigate the build-versus-cloud spectrum with a practical decision framework
- Measure what matters so you can improve performance intentionally

With these system-level best practices in place—and the technical grounding from earlier chapters—you're ready to move into optimization, where we turn architecture into concrete latency, throughput, and cost improvements.

Model Serving in an Agentic World

In this section, we use a sample agent to demonstrate how you can employ LLMs to create agentic user experiences, where software and services operate both interactively and autonomously.

While earlier chapters focused on serving models as standalone prediction engines, modern LLM applications increasingly embed models inside agentic systems. In these systems, the model is no longer invoked just once per request. Instead, it may be called repeatedly within a control loop that retrieves information, reasons over intermediate results, executes tools, and refines outputs before returning a final answer. This shift fundamentally changes the demands placed on model serving systems.

Agents are both powerful and complex. A single agent can perform sophisticated tasks, but in practice, agents are often organized into an agent network or platform, where multiple agents build on top of one another, collaborate, or specialize in different functions. This layered architecture enables remarkable autonomy but also introduces significant design and operational challenges—particularly for model serving.

For example, a single user interaction may trigger multiple LLM invocations, longer context windows, retrieval operations (RAG), or memory reuse (CAG). These behaviors increase token usage, amplify latency across chained calls, and create more dynamic traffic patterns. As a result, serving is no longer just about running a model efficiently—it must support orchestration, memory management, and system-level coordination.

To help you navigate model serving in the context of agent development, we focus here on the fundamentals. Using a bare-bones agent as an example, we illustrate how model serving supports the core mechanics of agent autonomy. With this foundation in place, you will be better equipped to explore and understand more advanced and multiagent systems on your own.

Defining Agents

Here we define an *agent* as an autonomous, LLM-powered system capable of:

- Understanding high-level goals
- Reasoning about how to achieve them
- Selecting and invoking external tools or data sources
- Adapting and iterating based on intermediate results
- Ultimately producing final outcomes with minimal or no human intervention

Types of agents include:

Research
: Reads academic papers, extracts key findings, and synthesizes a literature review

Coding
: Generates, reviews, tests, and debugs code, while managing deployment from a code repository

Business operations
: Automates data entry, generates reports, and distributes summaries to stakeholders

The fundamental difference between an LLM-powered agent and a traditional assistant program (such as a rule-based chatbot with precoded workflows) lies in autonomy. Traditional systems depend on predefined instructions from developers or step-by-step directions from users. In contrast, LLM-powered agents can execute complex workflows independently, thanks to their ability to:

- Interpret natural-language instructions
- Reason and plan actions
- Select and use tools
- Adapt to feedback from the environment or humans
- Maintain awareness of context and user preferences

In this section, our focus is on building your intuition for how LLM model serving underpins agentic applications. For readers interested in the design and implementation of single- or multiagent systems, we recommend the following books: *Building Applications with AI Agents* by Michael Albada (O'Reilly, 2025) and *AI Agents in Action* by Micheal Lanham (Manning, 2025).

A Sample Knowledge Agent

In this section, we use a real example to illustrate how model serving can power a *knowledge agent* designed to query and analyze information from PDF files. Although this sample agent is intentionally simple, this example highlights a common architectural pattern for building agents on top of model serving. For the sake of simplicity, we'll name this knowledge agent "Knowledge Agent."

To maximize portability, Knowledge Agent avoids local model hosting and databases. Instead, it relies on OpenAI's APIs for both LLM inference and embeddings, while storing all information in memory. The source code and setup instructions are available in the *KnowledgeAgent* folder of this book's GitHub repository. Before you read the rest of this chapter, please take some time to read these instructions and set this Knowledge Agent up on your own machine so that you can follow along with our examples. To gain a deeper understanding, we encourage you to run the sample agent yourself and experiment with different types of questions.

The sample Knowledge Agent supports document querying, intelligent planning for complex questions, summarization, and document analysis. Its knowledge base is built from the PDF files stored in the local *knowledge_files* folder. You can add your own PDFs to this folder, and the agent will automatically process them at startup:

```
~/llm-model-serving/ch04/KnowledgeAgent/ ls knowledge_files
  5-Level Paging and 5-Level EPT.pdf
  A Brief Introduction to the SAL.pdf
  A Brief Tutorial on Database Queries.pdf
  .. .. ..
```

Once the agent is running, you can ask it a wide range of questions. These may be direct information queries, such as:

- "What is 5-level paging and how does it work?"
- "What are Patricia tries and how are they optimized?"

They might also be advanced summarization and analysis tasks, such as:

- "How do these technologies relate to modern computing systems?"
- "Create a detailed comparison between database query optimization and data structure optimization."

The following example demonstrates the agent running locally:

```
~/llm-model-serving/ch04/KnowledgeAgent/ python agent.py
💬 Your question: What are the main types of database queries
   discussed in the tutorial?
✅ Response:
This tutorial by Lutz Hamel explores the key tools used in modern
relational database systems: database queries, data mining, and OLAP
   .. ... ..
```

Additional examples can be found in the Example Queries section of the README.

The Agent's Design

Let's begin by examining the structure of the Knowledge Agent. Figure 4-1 provides a visual overview of its key components. The Knowledge Agent is powered by two OpenAI-hosted models: text-embedding-3-small (for embeddings) and gpt-4.1-nano (as the LLM). The text-embedding-3-small model converts text into numerical vector representations that capture semantic meaning. These embeddings enable tasks such as semantic search, retrieval, and content matching.

The gpt-4.1-nano model serves as the agent's reasoning and language generation engine. It interprets instructions, plans subsequent actions, and produces natural-language responses.

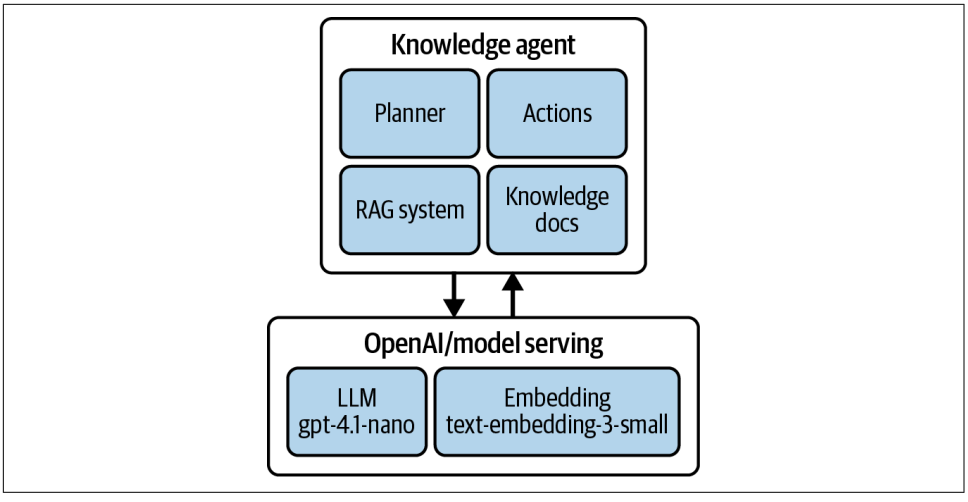

Figure 4-1. Knowledge Agent system overview

For more details on *embeddings* (converting text into numerical representations), see the OpenAI Embeddings User Guide.

On top of the models, the Knowledge Agent is built from these core components:

Knowledge Agent (orchestrator)
: The central controller that coordinates all components

Retrieval-augmented generation (RAG) system
: Processes PDFs, generates embeddings, and performs vector search

Planner
: Leverages the LLM to create intelligent execution plans for user queries

Actions (executor)
: Carries out specific tasks such as querying, summarization, and analysis

The Agent's Internal Workflow

Now, let's examine the internal logic of the Knowledge Agent. Figure 4-2 illustrates the workflow for handling a user query. Note that the "LLM serving" illustrated in the graph can be identical or distinct LLM models, depending on the action scenario.

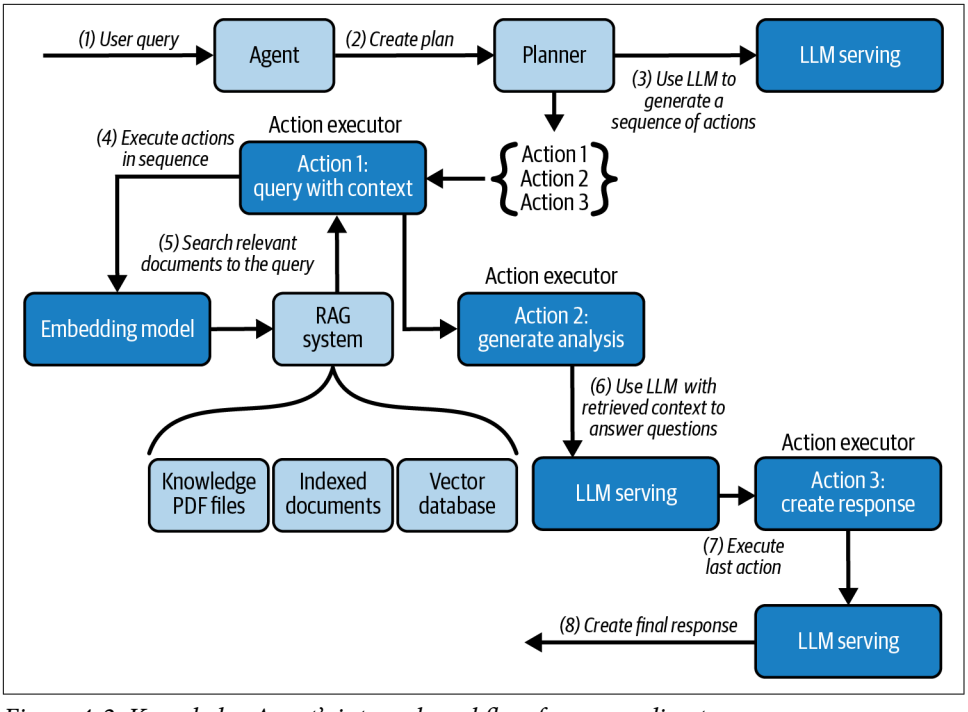

Figure 4-2. Knowledge Agent's internal workflow for responding to a user query

Let's walk through the workflow shown in Figure 4-2 step by step:

1. A user sends a question: for example, "Create a detailed comparison between database query optimization and data structure optimization."
2. The agent first calls its Planner component to design an execution plan.
3. The Planner calls the LLM, which generates a sequence of actions tailored to the query and the agent's available capabilities. For example, for the input from step 1, the LLM will create a three-step sequential plan: `query_rag_with_context`, `generate_analysis`, and `generate_summary`:

```
{
'plan': ['query_rag_with_context', 'generate_analysis', 'generate_summary'],
'reasoning': 'First, retrieving relevant contextual information ensures a
comprehensive understanding of both topics. Then, generating an analysis
allows for a detailed comparison highlighting differences and similarities.
Finally, summarizing consolidates the key points for clarity. This sequence
ensures a thorough and well-structured comparison.',
'estimated_steps': 3
}
```

4. Next, the ActionExecutor executes the planned actions (from step 3) in sequence; it will first work on the action `query_rag_with_context`. You can find the implementation of all three actions in our code repo, in the *actions.py* file.
5. The `query_rag_with_context` action first triggers the RAG system to locate documents relevant to the query from the knowledge base, then uses the LLM to summarize the documents. The RAG system functions as a semantic search engine over indexed documents. A sample relevant doc returned by the RAG system looks like following:

```
Document 1 (Source: A Brief Tutorial on Database Queries, Data Mining, and
OLAP (hamel-197-manuscript-final).pdf):\nA Brief Tutorial on Database
Queries, Data Mining, and OLAP  Lutz Hamel Department of Computer Science
and Statistics University of Rhode  Island Tyler Hall Kingston ...
```

6. The ActionExecutor executes the second action, `generate_analysis`. This action sends the user query and the documents from step 5 as context to the LLM.
7. The LLM produces a deeper comparative analysis based on the query and its context.
8. Finally, the ActionExecutor works on the last action, `generate_summary`. It calls the LLM to summarize the analysis from the previous step.
9. The agent consolidates the outputs from all actions and returns the final result to the user.

By walking through this workflow, you can see how the Knowledge Agent decomposes a complex query into steps, retrieves the relevant information, and synthesizes an autonomous response with the help of an LLM.

Agent Autonomy

The sample Knowledge Agent demonstrates that LLM-powered agents differ fundamentally from conventional applications, which typically work through *procedural execution*: they execute commands through fixed logic and predefined workflows. Instead, an agent can operate with *goal-driven autonomy:* it can interpret the user's intent, choose an approach, integrate multiple tools, and deliver results, all with minimal user intervention.

Defining actions

To enable autonomy, an agent defines a set of reusable *actions*—discrete operations that it can execute in response to a plan. Typical actions include summarizing documents, answering queries, and performing deeper analysis. In our sample agent, each action is implemented as an LLM call with a specialized prompt template. For

example, the `create_analysis_prompt` action generates a structured analysis based on the user's query and supporting context:

```
    def create_analysis_prompt(self, query: str, context: str) -> str:
        """Create a prompt for detailed analysis."""
        prompt = f"""
    You are an expert analyst. Please provide a detailed
    analysis of the following question based on the
    provided context.

    Question: {query}

    Context:
    {context}

    Please provide:
    1. A comprehensive analysis
    2. Key insights and findings
    3. Relevant examples from the context
    4. Any limitations or gaps in the available information

    Analysis:
    """
        return prompt
```

In practice, agent actions are not limited to LLM prompts. Depending on the design, they may involve:

Tool calls
: Running a web search, querying an API, or executing a database command

System operations
: Managing files, triggering workflows, or interacting with external services

Reasoning steps
: Intermediate computations or planning subroutines that do not directly involve a user-facing response

These diverse actions allow an agent to combine reasoning, knowledge retrieval, and external capabilities into a coherent workflow.

Model Context Protocol (MCP)

One popular approach to enabling tool use in agents is the *Model Context Protocol* (MCP). MCP standardizes how LLMs interact with external tools and data sources. Instead of hardcoding integrations, MCP defines a consistent interface for the agent to discover tools available in its environment, call them with structured inputs, and incorporate the results back into its reasoning process.

This protocol simplifies development and improves reliability by decoupling tool definitions from the core agent logic. MCP also helps address the brittleness of ad-hoc prompt engineering by giving LLMs a predictable way to invoke tools. You can check out the book *AI Agents with MCP* by Kyle Stratis (O'Reilly, 2026) for more detail.

For clarity and focus, our sample agent keeps its actions simple—sticking primarily to LLM-based calls with prompt templates. While real-world agents often combine LLM reasoning with tool invocation through protocols like MCP, this stripped-down design highlights the fundamentals of how model serving supports autonomy.

Planning with LLMs

Rather than relying on predefined workflows, the agent's Planner uses the LLM to interpret a high-level instruction, break it down into subtasks, and determine the optimal execution path. The agent orchestrates these steps automatically, freeing users from manual micromanagement, such as parsing PDFs, locating relevant sections, and summarizing results.

A simplified version of the planning logic is shown here:

```
def create_plan(self, query: str) -> Dict[str, Any]:
  """Create an execution plan for the given query using OpenAI."""
  logger.info(f"Creating plan for query: {query}")

  # Create planning prompt
  planning_prompt = self.llm_manager.create_planning_prompt( \
     query, self.available_actions)

  # Get plan from OpenAI
  plan_response = self.llm_manager.generate_response( \
     planning_prompt, temperature=0.3)

  # Parse JSON response
  plan = self._parse_plan_response(plan_response)
  logger.info(f"Created plan: {plan}")
  return plan
```

Retrieval-Augmented Generation (RAG)

In the previous section, we introduced the RAG system (Figure 4-2), which serves as a search engine for the agent to locate text relevant to a user query within its knowledge base. Since RAG is central to how agents operate, we will explore it in more detail here.

Why RAG?

Why do LLM-powered agents need an auxiliary system like RAG? LLMs, while powerful, have well-known limitations:

- Their knowledge is limited and static, since their training data is frozen at a specific cutoff date.
- They may *hallucinate*, or generate content that is fluent but factually incorrect.
- They often lack specialized or up-to-date information, and thus have domain-specific gaps in their knowledge.

RAG mitigates these issues by augmenting the LLM with external knowledge retrieved at query time. Instead of relying solely on the model's internal training data, RAG enables responses enriched with domain-specific and continuously updated information. This improves accuracy, reliability, and adaptability—particularly for knowledge-intensive tasks. As a result, RAG-equipped agents are far more reliable and adaptable than agents that rely on LLM generation alone. Figure 4-3 illustrates the two main workflows of a basic RAG system.

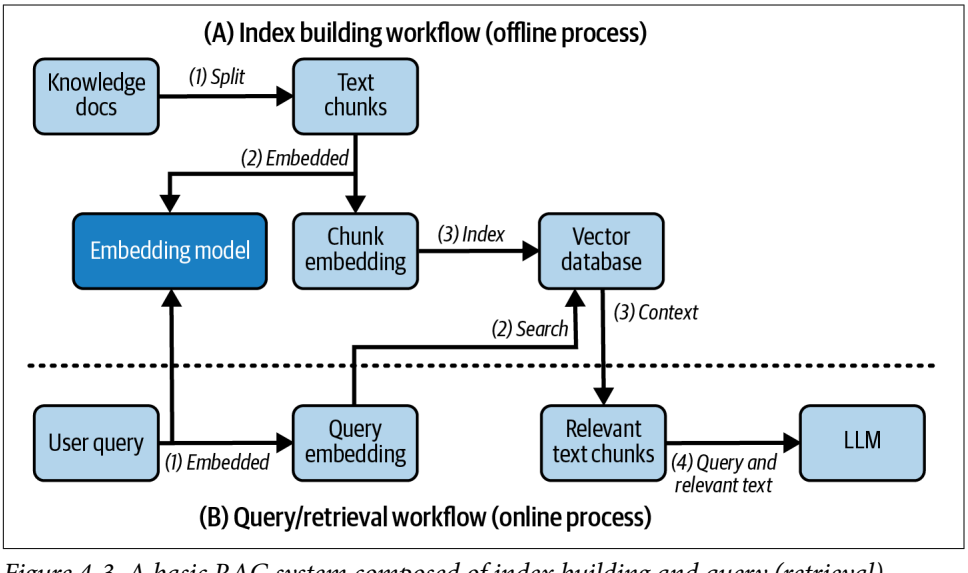

Figure 4-3. A basic RAG system composed of index building and query (retrieval) workflows

Index-building workflow (offline process)

The index-building workflow (part A in Figure 4-3), is an offline data process, typically scheduled to run periodically. It begins by cleaning and parsing raw knowledge documents—often in formats such as HTML or PDF—into plain text. This text is then segmented into chunks, typically around 1,000 tokens each.

Chunking (splitting large content into smaller chunks) is more than just a preprocessing step for embeddings. It defines the *granularity of retrieval* (how much text the system can consider at once) and serves as the stage where *offline batch inference* occurs. Instead of embedding entire documents on demand, the system precomputes embeddings for each chunk in bulk, making query-time retrieval much faster and more efficient.

Once chunking is complete, each segment is encoded into a dense vector representation using an embedding model. These chunk embeddings are then indexed and stored in a vector database, where they become the foundation for efficient similarity search during the online query workflow.

Fine (Small) Versus Coarse (Large) Chunking

Smaller chunks provide higher precision, since retrieval focuses on narrowly relevant passages. However, they can lose context if split too aggressively. Larger chunks provide richer context but may dilute precision by including unrelated content.

In practice, chunk size is a trade-off, and the optimal balance depends on the application domain and the LLM's context window.

The query/retrieval workflow (online process)

The query workflow (part B in Figure 4-3), is an online process that happens in real time when customers interact with the agent. When an agent receives a query, it first gets the query text's embedding vector by using the embedding model. Then it searches for the vectors in the database that are closest to the query vector (using measures like cosine similarity). The agent retrieves the most relevant document chunks from the vector database by using the selected vectors and ends by sending them to the LLM with the user query to generate an answer.

While Figure 4-3 shows a straightforward, "naive" RAG pipeline, production-grade RAG systems introduce additional complexity—such as chunking, ranking, deduplication, and multisource retrieval. For a deeper exploration of RAG, we recommend you start from Gao et al.'s 2023 paper, "Retrieval-Augmented Generation for Large Language Models: A Survey".

Why Split Text into Small Chunks?

LLMs have a limit on the maximum number of tokens they can process at once, including input and output tokens. This means there is limited space for context. That space is known as the *context window*. To accommodate these limitations, RAG segments the text it extracted into smaller, digestible chunks, so only a few highly relevant text chunks are sent to the LLM as context.

Cache-Augmented Generation (CAG)

In the previous section, you saw how RAG enhances language models by integrating external knowledge sources. While powerful, RAG also introduces several challenges.

There are extra steps and latency before calling the LLM. And when selecting relevant documents with limited tokens, RAG agents may select incorrectly. There is also additional complexity involved in building and maintaining a complicated search system.

As LLMs evolve, their context windows continue to expand. For instance, the Claude Sonnet 4 model (as of August 2025) supports a 1-million-token context window. With such capacity, the need to carefully select only a handful of "most relevant" documents diminishes. Instead, an entire knowledge base—or large portions of it—can be cached inside the model's working context. This shift makes it possible to overcome LLMs' key limitations (like static knowledge, hallucinations, and domain-specific gaps) without relying on a complex RAG system.

Cache-augmented generation (CAG) leverages the extended context windows of modern LLMs by preloading knowledge directly into the model's KV cache. Rather than performing retrieval at query time, CAG loads relevant resources in advance so that the model can generate answers during inference using the cached context. This approach eliminates retrieval latency and reduces system complexity, while still enabling the LLM to ground its responses in external knowledge. Figure 4-4 compares RAG and CAG systems.

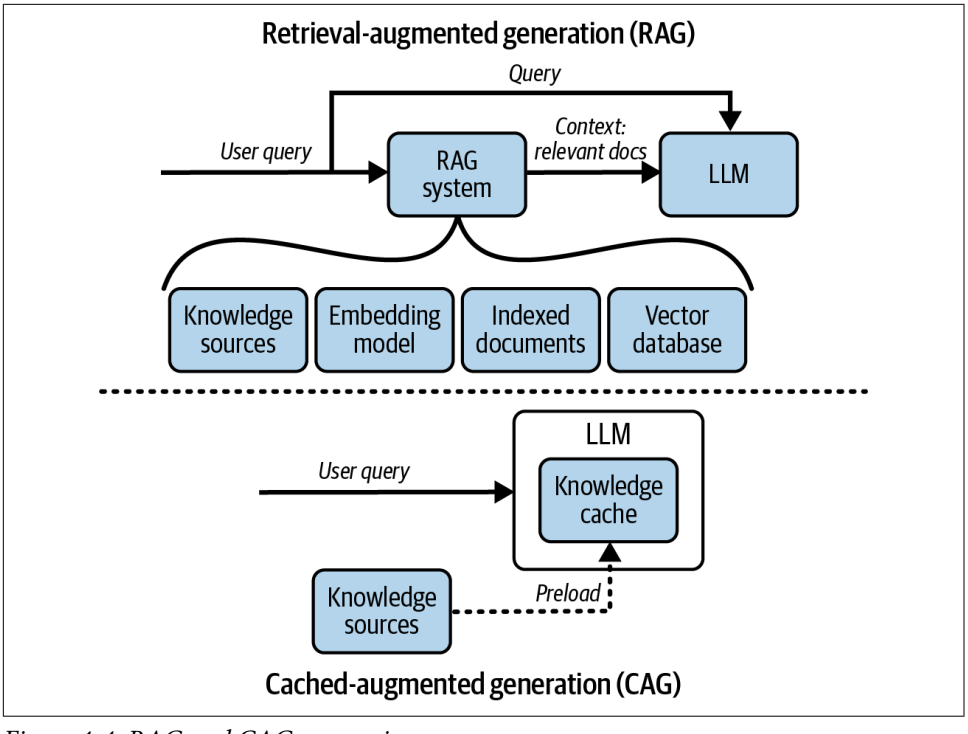

Figure 4-4. RAG and CAG comparison

In RAG, every query triggers a search process to find context dynamically, while in CAG, once the knowledge docs are loaded into the LLM's cache, queries are answered *directly* using the external knowledge in the context. This improvement greatly simplifies the agent design, as shown in Figure 4-5.

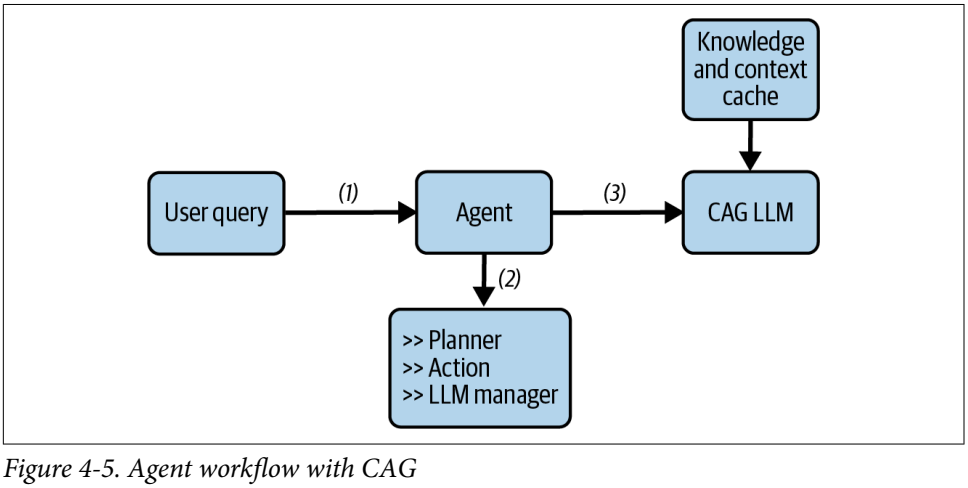

Figure 4-5. Agent workflow with CAG

Compare the agent design in Figure 4-5 to the one in Figure 4-2. CAG greatly simplifies the system architecture by removing the RAG component entirely. Instead of managing embeddings, indexes, and vector databases, the agent interacts directly with a CAG-enabled LLM already loaded with the knowledge and context. For a deeper dive into this paradigm, we recommend Chan et al.'s 2024 paper, "Don't Do RAG: When Cache-Augmented Generation Is All You Need for Knowledge Tasks".

While CAG is promising, it also introduces its own challenges—particularly in terms of performance and efficiency. Large context windows and cache management can increase memory and compute requirements. In Chapter 7, we will explore techniques to optimize CAG model serving, with a focus on leveraging the LLM cache to improve efficiency.

RAG Versus CAG

RAG and CAG are not competing techniques—they solve different problems at different layers of the system. RAG improves answer quality by retrieving relevant external knowledge and expanding the prompt context. CAG improves serving efficiency by reusing previously computed context and reducing redundant KV-cache computation.

If your primary concern is knowledge grounding and freshness, start with RAG. If your primary concern is latency, throughput, and cost efficiency, introduce CAG. In production LLM systems—especially agentic workflows—it is common to use both: RAG to enrich the input and CAG to optimize execution.

How Agents Use Model Serving

As you saw from the Knowledge Agent example, autonomous agents often require orchestrating multiple models and tools to complete tasks. Typical components include:

- LLMs for reasoning, planning, and dialogue generation
- Embedding models for retrieval, similarity search, and semantic matching
- Vision or speech models for multimodal perception
- Tool-specific models for tasks such as code generation, classification, and summarization
- External tools such as APIs, databases, or services, which the agent can invoke as needed

A key capability of modern agents is tool calling. Tool calling is a four-step process:

1. The LLM reasons over the user's request and the available tools.
2. It generates structured output (often in JSON) that encodes the required inputs for the selected tool.
3. The agent executes the tool call.
4. The results are passed back into the LLM, which iterates until the task is complete.

This pattern allows agents to extend beyond pure text generation, combining reasoning with precise tool execution.

All of these models and tools are typically accessed through model serving services (such as HTTP or gRPC APIs), so the agent can invoke them on demand. Since agents operate interactively and often in real time, high-performance, low-latency, and cost-efficient serving is critical to the success of any agent application.

Now that you've seen how models and tools are used in agentic applications, the remainder of this chapter will explore different approaches to building a production-grade model serving system.

LLM Serving in Enterprise Systems: An Overview

In this section, we introduce an abstract layered architecture that has been adopted by leading providers, such as OpenAI, Anthropic, and Together AI, as well as large enterprises, when deploying LLMs at cloud scale.

Compared with the simplified serving examples in earlier chapters, real-world deployments involve far more than simply hosting and executing models. At scale, providers must also address requirements such as authentication, pricing, resource management, networking, optimization, experimentation, observability, and on-call support. Meeting these business and operational demands requires multidisciplinary engineering teams of hundreds or even thousands of engineers.

Architecting such a system is challenging. The serving platform must enable many teams, each with distinct responsibilities, to contribute to a single evolving system without creating bottlenecks or excessive interdependencies. It must balance fast iteration with stability, all while operating under the constraints of cost, reliability, and user experience.

Figure 4-6 presents what a layered enterprise model serving architecture looks like in practice. This conceptual view highlights the distinct requirements and challenges that arise at each layer.

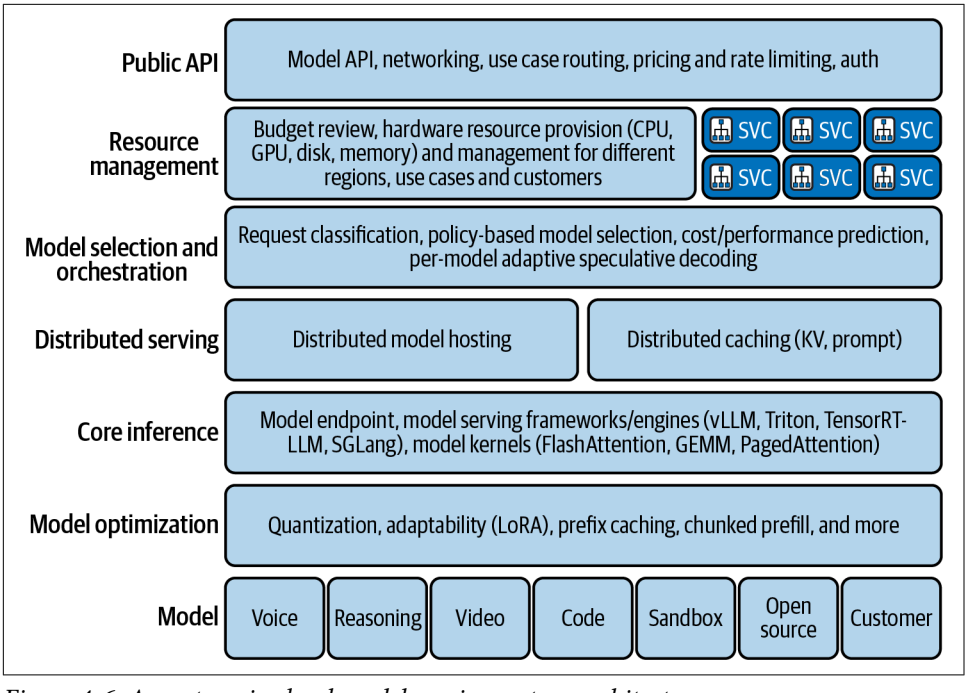

Figure 4-6. An enterprise-level model serving system architecture

Let's walk through the layers of Figure 4-6 from top to bottom. At each layer, we'll examine its purpose and key challenges.

Public API Layer

The public API is the primary external interface through which customers, developers, and internal services interact with the LLM system. It manages networking, authentication, pricing, rate limiting, and request routing.

Key challenges include:

High concurrency
: This layer needs to support millions of simultaneous client connections.

Fair usage and monetization
: This layer must enforce quotas, prevent abuse, and support accurate billing.

Low-latency global access
: It uses geo-routing and regional caching to serve requests from the nearest region.

Security
: This layer must ensure authentication, tenant isolation, and protection against network attacks.

Resource Management Layer

This layer manages the infrastructure hardware resources (such as CPU, GPU, memory, disk, and networking) required to serve models, use cases, and customer workloads across regions. Budget reviews and cost allocation are often integrated here.

Key challenges include:

Capacity planning
: This layer must forecast demand while avoiding costly overprovisioning.

GPU utilization
: This layer must manage heterogeneous GPU pools across datacenters and maintain high utilization.

Customer prioritization
: This layer must enforce quotas and reservations for critical workloads, while enabling preemptible scheduling for lower-priority tasks.

Model Selection and Orchestration Layer

The orchestration layer determines which model(s) to use for each request, balancing accuracy, latency, and cost. Beyond selecting a single model, this layer can orchestrate multiple models in tandem, for example by applying speculative decoding or routing between model families.

Speculative Decoding

Speculative decoding accelerates inference by having a smaller "draft" model predict several tokens ahead, while a larger "target" model verifies them in parallel. If the draft is correct, the tokens are accepted; otherwise, corrections are applied. This technique reduces token-by-token latency without degrading quality. We will explore speculative decoding in more detail in Chapter 7, and you can also refer to the paper "Fast Inference from Transformers via Speculative Decoding" (Leviathan, Kalman, and Matias, 2022).

Key challenges include:

Cost–quality trade-offs
: Not all tasks require the largest models. For example, OpenAI often defaults to gpt-4o-mini unless the user explicitly requests a stronger model. Also, triaging which model to use means this layer must understand the query.

Load balancing
: This layer must prevent bottlenecks by distributing traffic across model pools.

Latency-sensitive use cases
: For these cases, this layer can employ smaller or faster models or advanced optimizations such as speculative decoding.

Distributed Serving Layer

This layer sets up the distributed infrastructure for LLM model execution. This generally falls into two groups: distributed model hosting for large models and distributed caching (such as KV caching, prompt caching, and semantic caching) to reduce unnecessary computation. Here are some challenges addressed in this layer:

Hardware limitations
: LLM sizes often exceed single-GPU memory.

Multi-GPU and multi-node coordination
: Multi-GPU and multi-node distributed models must efficiently share the request's context to maintain the serving service-level agreement (SLA).

Caching for efficiency
: This layer must route traffic based on cache content, such as KV cache-aware routing, and reduce redundant inference for repeated or similar inputs.

Core Inference Layer

This is where models are actually executed. It integrates model serving frameworks (vLLM, Triton, TensorRT-LLM, SGLang) with optimized kernels (FlashAttention, GEMM, PagedAttention) and exposes them as web endpoints. We explored this layer in detail in Chapter 3.

Model Optimization Layer

This layer focuses on applying all kinds of model optimization techniques to improve the model's performance and efficiency without retraining from scratch. We will discuss different optimization techniques and their trade-offs in detail in Chapters 5, 6, 7, and 9.

Model Layer

The bottom layer provides the actual trained models to the serving system and it:

- Moves models from internal training pipelines or external sources into production
- Organizes models by capability (voice, reasoning, video) and purpose (sandbox, experimentation, production)
- Handles the tracking and versioning of models as the model evolves

As the field advances, new models and technologies will reshape system design. However, the layered architecture pattern—separating concerns such as API, orchestration, inference, and optimization—remains essential. It allows teams with different responsibilities to innovate and optimize independently while contributing to a cohesive serving platform.

Building with an Open Source Stack

In this section, we discuss how to implement an enterprise model serving system using a set of our favorite open source software components. The goal is not to prescribe a single solution, but to illustrate practical design choices and give you a starting point for building your own serving platform.

First, Figure 4-7 provides a quick system overview.

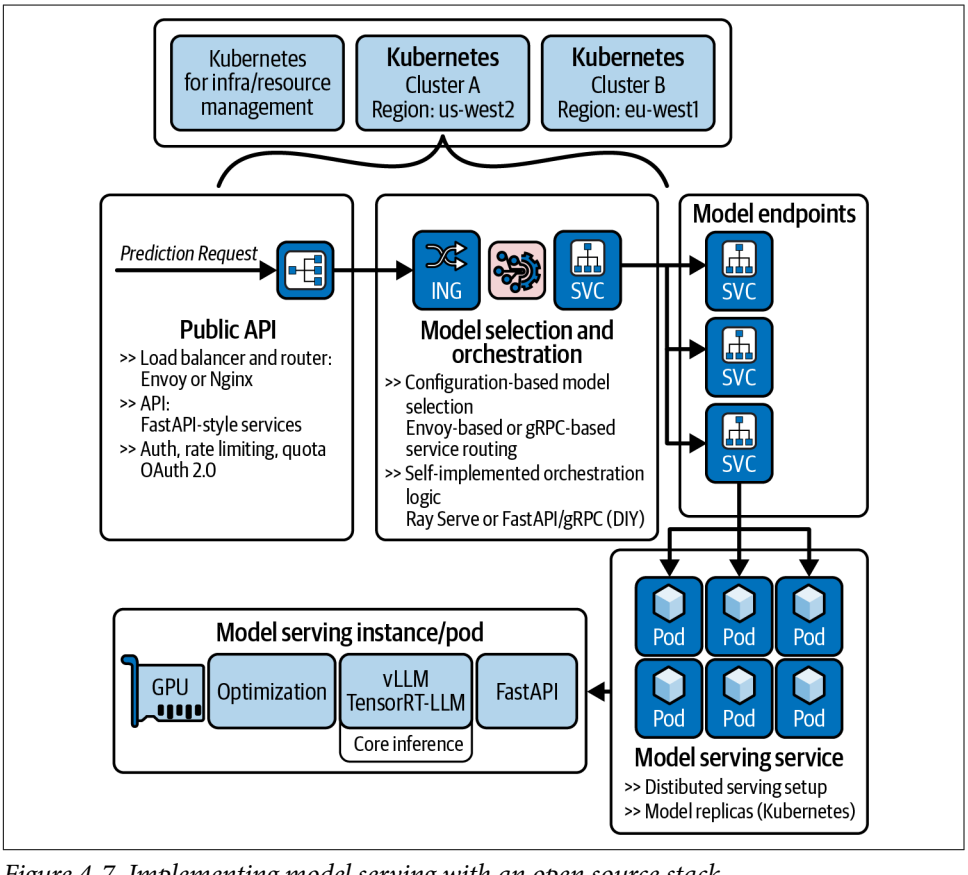

Figure 4-7. Implementing model serving with an open source stack

In this design, we chose to build an entire serving stack on top of Kubernetes, an open source platform for automating the deployment, scaling, and management of containerized applications. Beyond its core features, Kubernetes also has a large open source ecosystem built on top of it, encompassing functions such as metrics, logging, networking, authorization, and hardware management. Because of these, Kubernetes has been widely adopted and is the backbone for many successful cloud providers (Amazon AWS, Google GCP, and Microsoft Azure) and foundation model vendors (OpenAI, Anthropic).

In Figure 4-7, we utilize Kubernetes and its ecosystems to take care of resource management, networking, traffic routing and load balancing, service hosting and scaling, and service metrics and monitoring. To learn more about Kubernetes, we recommend the books *Kubernetes: Up and Running* by Brendan Burns et al. (O'Reilly, 2022) and *Kubernetes Best Practices* by Brendan Burns et al. (O'Reilly, 2023).

Implementing Public API

For our public API example, we decided to use FastAPI and Kubernetes to implement the web API, quota limits, auth, and tenant management. Let's look at a few code examples.

First, using FastAPI, we built a sample web service (`enterprise-model-api`) that exposes a chat API (`/v1/chat/completions`). We also leveraged FastAPI's dependency inject function (`Depends`) to attach authorization logic to this API:

```
@app.post("/v1/chat/completions")
async def chat(req: ChatReq, idp=Depends(require_auth)):
    await rate_limit(idp["tenant"])
    # select model, route traffic based on tenant information
```

We had the option of using either a JWT token or an API key to authenticate and authorize user requests. During this process, the platform also retrieves customer metadata, which is essential for downstream steps such as quota enforcement, traffic routing, and model selection:

```
# authorize request either with JWT or api_key
async def require_auth(
   api_key=Depends(verify_api_key),
   claims=Depends(verify_jwt)):
    if not api_key and not claims:
        raise HTTPException(401, "Missing API key or JWT")
    tenant = claims.get("tenant") if claims else await \
        rds.hget(f"keys:{api_key}", "tenant")
    if not tenant: raise HTTPException(403, "Unknown tenant")
    return {"tenant": tenant, "claims": claims, "api_key": api_key}

async def verify_jwt(authorization: str):
    typ, token = authorization.split(" ", 1)
    if typ.lower() != "bearer":
```

```
            raise HTTPException(401, "Bad auth type")

        # verify JWT and return token content
        .. .. ..
        return jwt.decode(token, \
           jwt.algorithms.RSAAlgorithm.from_jwk(json.dumps(key)), \
           algorithms=[headers["alg"]], audience="api://llm", \
           options={"verify_exp": True})

    def verify_api_key(x_api_key: str | None = Header(default=None)):
        ... ...
```

For autoscaling, we leveraged Kubernetes. In the following configuration, if the `enterprise-model-api` service's average CPU usage exceeds 70%, Kubernetes will trigger horizontal scaling to add 3 to 15 more instances to this service:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata: { name: enterprise-model-api-hpa }
spec:
  scaleTargetRef: { apiVersion: apps/v1,
      kind: Deployment, name: enterprise-model-api }
  minReplicas: 3
  maxReplicas: 15
  metrics:
  - type: Resource
    resource: { name: cpu, target: { type: Utilization,
      averageUtilization: 70 } }
```

For rate-limiting requests to the public API service, Nginx and Envoy are the two most common approaches to use as ingress proxies. The following example shows a simple Kubernetes configuration using Nginx to limit traffic to the `enterprise-model-api` service:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: api-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    # rate limit the traffic to 50 request per second max
    nginx.ingress.kubernetes.io/limit-rps: "50"
    nginx.ingress.kubernetes.io/limit-burst-multiplier: "5"
    nginx.ingress.kubernetes.io/proxy-body-size: "8m"
spec:
  rules:
  - host: api.yourorg.example
    http:
      paths:
      - path: /
        pathType: Prefix
        backend: {service: { name: enterprise-model-api, port: { number: 80 } } }
```

If you prefer Envoy, refer to the tutorial "Enabling Rate Limits Using Envoy" for step-by-step guidance. Both Nginx and Envoy integrate seamlessly with Kubernetes and can be enabled through configuration, highlighting Kubernetes' strength and flexibility as a serving platform.

Implementing Model Selection

After the public API, let's look at the model selection phase. Depending on the requirements, we can implement the model selection logic in a public API, in a separate service (middleware), or simply in a static routing config.

In our simplified example, the model selection logic is implemented within the public API service's chat endpoint (`/v1/chat/completions`). Based on the token size of the incoming request, the system does one of two things:

- Directly forwards the request to the selected model backend
- Uses speculative decoding, where a faster, cheaper model generates multiple future tokens, and a slower but more accurate model verifies them in bulk. This approach avoids redundant step-by-step generation and provides significant latency improvements. (We will explore speculative decoding in greater detail in Chapter 7.)

Here is the logic:

```
@app.post("/v1/chat/completions")
async def chat(req: ChatReq, idp=Depends(require_auth)):
    await rate_limit(idp["tenant"])
    # choose the right model for the given request
    ep = choose_endpoint(req.model, idp["tenant"])

    # basic classifier: use speculation for long outputs, else direct
    if req.model.draft_enabled and req.max_new_tokens > 1024:
        draft_ep = config.get_draft_endpoint(req.model)
        # use draft model for generation, target model for validation
        gen = speculative_decode(req,
                 endpoint_draft=draft_ep,
                 endpoint_target=ep)
    else: # simple pass-through stream
         gen = passthrough(ep)
```

Speculative Decoding Has Many Forms of Implementation

Speculative decoding has multiple variations, such as per-model adaptive speculative decoding and multi-draft decoding. While they all share the same foundational idea—using a smaller, faster model to reduce latency—they can be implemented at different layers of the architecture shown in Figure 4-6, depending on your specific optimization goals.

Now, let's look at an example of self-implemented model endpoint-selection logic. The system first applies general preferences—such as region, cost, and load-balancing considerations—then layers on tenant-specific (customer) requirements to customize routing for individual clients:

```
def choose_endpoint(model: str, tenant: str):
    cfg = load_routes()
    # policy example: tenant allow-list, cost class, region, canary
    route = cfg["models"].get(model) or cfg["aliases"].get(model)
    if not route: raise HTTPException(404, f"Unknown model {model}")
    # weighted canary
    if "canary" in route and random() < float(route["canary"]["weight"]):
        return route["canary"]["url"]
    # tenant override
    route_over = (route.get("tenants") or {}).get(tenant)
    if route_over: return route_over["url"]
    return route["url"]
```

Implementing a Model Serving Endpoint

At the model hosting layer, this is where single-model and multi-model serving come into play. To implement this yourself, you can follow the designs introduced earlier in Figures 3-7 and 3-8, and build with tools such as vLLM, NVIDIA Triton, Kubernetes, and FastAPI.

If managing scalability and networking for serving instances feels too heavy, you can instead use Ray Serve—a scalable, framework-agnostic model serving library—for building online inference APIs. In the next few sections, we will demonstrate how to use Ray Serve to implement single-model and multi-model hosting.

Single-model hosting on Ray

First, create a Python class called `QwenVLLM` to contain the serving logic. Let's start by initializing the model (Qwen3) in vLLM:

```
class QwenVLLM:
    def __init__(self):
        # Initialize vLLM async engine once per replica
        args = AsyncEngineArgs(
            model="Qwen3-Thinking-2507",
            tensor_parallel_size=1,
            trust_remote_code=True,
            max_num_batched_tokens=4096,
            gpu_memory_utilization=0.9,
        )
        self.engine = AsyncLLMEngine.from_engine_args(args)
```

Then define the model serving request's entry point and execution logic:

```
class QwenVLLM:
    # entry point for the generation/prediction request.
    async def __call__(self, http_request: Request) -> str:
        """
        Accepts a raw JSON body:
          { "text": "Explain KV cache simply.",
         "max_new_tokens": 200, "temperature": 0.7 }
        Returns the generated string.
        """
        body = await http_request.json()
        text = body["text"]
        max_new = body.get("max_new_tokens", 256)
        temp = body.get("temperature", 0.7)
        return await self.generate_text(text, \
            max_new_tokens=max_new, temperature=temp)

      async def generate_text(self, text: str, \
          max_new_tokens: int = 256, \
          temperature: float = 0.7) -> str:
          ... ...
        # run vLLM engine to execute Qwen model
        async for out in self.engine.generate(prompt, \
            params, request_id=req_id):
            # For vLLM with stream=True, incremental delta is here:
            delta = out.outputs[0].text_delta
            if delta:
                chunks.append(delta)
        return "".join(chunks).strip()
```

After implementing the model serving code, you can host it in Ray Serve by adding a decorator `@serve.deployment` to the `QwenVLLM` class. With this declaration in place, during service deployment, Ray will automatically provision compute resources and deploy the service for Qwen model serving.

The following code instructs Ray to serve the Qwen3 model with three service instances, each allocated a dedicated GPU:

```
# Create Ray Serve instance for hosting Qwen3 model
@serve.deployment(
    name="qwen_vllm",
    # declare the number of service instances for the model
    num_replicas=3,
    # needs a GPU for speed
    ray_actor_options={"num_cpus": 0.5, "num_gpus": 1},
)
class QwenVLLM:
 .. .. ..
```

Multi-model hosting on Ray

If you need to host a group of models with similar input types on a shared service, Ray Serve offers a feature called model multiplexing to serve multiple models from a pool of service instances (replicas).

By using the `serve.multiplexed` and `serve.get_multiplexed_model_id` APIs, you can extend your single-model hosting service to multi-model hosting in Ray. We'll show you a quick example for hosting three different sentiment models on Ray Serve with two service instances.

First, you'll define the model execution logic, which loads and executes the model in real time per your inference request:

```
@serve.deployment(
    num_replicas=2,  # two instances/replicas
    ray_actor_options={"num_cpus": 1} # resources per replica
)
class MultiModelService:
   # entry point for model inference requests
   async def __call__(self, request: Request):
       # Pick the model id for this request
       model_id = _serve.get_multiplexed_model_id()

       # load model and run model inference
       tokenizer, model = await self.load_model(model_id)
       inputs = tokenizer(text, return_tensors="pt")
       with torch.no_grad():
           logits = model(**inputs).logits
           probs = torch.softmax(logits, dim=-1).tolist()[0]

       return JSONResponse({
           "model_id": model_id,
           "label_probs": probs,  # [neg, pos] for SST-2 style models
       })
```

From the `__call__` function of the `MultiModelService` class, you can see that the multi-model inference code's implementation is very similar to that for single-model serving. The main difference is that, in single-model serving, you preload the model in the `__init__` function, while for multi-model hosting, you'll try to load the requested model in real time for every prediction request using `await self.load_model(model_id)`, since requests may ask for different models.

The drawback of this approach is that loading a model for every request introduces additional latency. To mitigate this, annotate the model loading function (`load_model`) with `@serve.multiplexed`. This tells Ray Serve to apply its caching mechanism when possible and to manage model swapping when resources such as memory are constrained.

The following examples illustrate how to enhance the model-loading logic with multiplexing enabled:

```
# Example registry: map model_id -> HF repo
# This model source could be S3 or any other storage
MODEL_REGISTRY = {
   "distilbert-sst2": "distilbert-base-uncased-finetuned-sst-2-english",
   "bert-sst2": "textattack/bert-base-uncased-SST-2",
   "roberta-sst2": "roberta-base-openai-detector",
}

@serve.deployment(ray_actor_options={"num_cpus": 1})
class MultiModelService:

   # enable multi model support and
   # set model replica count for each model
   @serve.multiplexed(max_num_models_per_replica=2)
   async def load_model(self, model_id: str) \
            -> Tuple[AutoTokenizer, torch.nn.Module]:
       """
       Lazily load & cache a model per model_id on this replica.
       Ray Serve will LRU-evict when cached models
         exceed max_num_models_per_replica.
       """
       repo = MODEL_REGISTRY[model_id]

       tokenizer = await asyncio.to_thread(\
              AutoTokenizer.from_pretrained, repo)
       model = await asyncio.to_thread(\
              AutoModelForSequenceClassification.from_pretrained,\
              repo)
       model.eval()
       return tokenizer, model
```

You can use a HTTP header `ray_serve_multiplexed_model_id` to specify the ID of the model to call in the prediction request. Ray will handle traffic routing, load balancing, and resource management on the backend. For example, the following curl request sends a sentiment prediction request to the `distilbert-sst2` model host in Ray Serve:

```
curl -s -X POST "http://127.0.0.1:8000/SentimentService" \
  -H 'content-type: application/json' \
  -H 'ray_serve_multiplexed_model_id: distilbert-sst2' \
  -d '{"text":"I absolutely loved this!"}'
```

Building Your Own Serving Stack Is Not Difficult

While the system may appear to involve many moving pieces, most of the components we've introduced in this chapter have already been battle-tested in production environments. By leveraging Kubernetes for infrastructure management, Ray Serve or Triton Inference Server for model hosting, and vLLM for distributed LLM serving and optimizations (like kernel-level acceleration and caching), you can assemble a robust, flexible serving stack.

With some lightweight custom logic in a self-implemented middleware microservice, it is entirely possible to build a full enterprise-grade serving solution yourself. Since the libraries and frameworks introduced here have matured over years of adoption, there are many strong reference implementations from which you can draw inspiration. In addition, AI coding assistants can significantly accelerate development by handling boilerplate and integration tasks.

Building with a Cloud Vendor

In this section, we use AWS SageMaker, Amazon's AI platform, to illustrate six different ways to build a model serving system on a public cloud.[1] We will walk you through these options, from the fully managed end of the spectrum to the most customizable approaches. The goal is not to teach you how to operate SageMaker step by step, but to help you understand the fundamental logic behind cloud vendors' serving options so you'll be better equipped to navigate the landscape and make informed choices, even when the variety of options feels overwhelming.

In this section, we use the terms *Docker image* and *Docker container* interchangeably, both referring specifically to the model serving service and its execution environment.

Option 1: Fully Managed Foundation-Model Serving

Amazon Bedrock is a fully managed service for serving foundation models via a simple API. Of the options we present here, it is the least customizable but the easiest to use. With Bedrock, you don't handle training or hosting the model yourself. You simply select a model Bedrock supports (such as Amazon Titan, Anthropic Claude, or Stability AI's Stable Diffusion), then make API calls to get predictions.

1 While the examples and options we list in this chapter are specific to AWS, their functionality is quite general. You can find corresponding components in other popular cloud vendors, such as Google GCP and Microsoft Azure.

Bedrock provides several foundation models—both from AWS and from third-party model providers such as Anthropic and Cohere—that you can use out-of-the-box, without having to deploy or manage any infrastructure. This is ideal for quickly integrating AI capabilities into applications or prototyping with minimal setup.

The price model of Bedrock is "pay-as-you-go," typically charging per request or per output token rather than, for instance, by the hour. This means you don't run any servers in your account for these models—it means that AWS manages the scaling and computing resources internally.

As a result, Bedrock offers zero DevOps overhead: there are no servers to provision or scale and no model containers to build. The trade-off is that you are limited to the models and configurations AWS provides, with relatively limited customizability. For example, you cannot change the model's architecture or training; you can only do moderate fine-tuning or prompt customization.

Using Bedrock just takes three steps:

1. Create an API key in your AWS account.
2. Select the model you want to use from Bedrock's model catalog.
3. Initialize a Bedrock client and submit API calls.

The following example illustrates this process:

```
# Set the API key as an environment variable
os.environ['AWS_BEARER_TOKEN_BEDROCK'] = "..."

# Create the Bedrock client
client = boto3.client(
   service_name="bedrock-runtime",
   region_name="us-west-2"
)

# Define the model and message
model_id = "us.anthropic.claude-3-5-sonnet-20240620-v1:0"
messages = [{"role": "user", "content": [
    {"text": "Hello! Can you tell me about Amazon Bedrock?"}
]}]

# Make the API call
response = client.converse(
   modelId=model_id,
   messages=messages,
)
```

Limitations

Under the hood, what Amazon Bedrock provides is a *serverless managed inference environment* for foundation models. You interact with Bedrock via a high-level API, either through AWS SDKs or HTTPS endpoints.

When you call the Bedrock API to invoke a model, AWS's infrastructure (which you do not see or manage) handles provisioning the GPU instances, loading the model weights, running the inference, and returning the result. This is conceptually similar to calling an external AI API (like OpenAI's)—but it's an AWS service call within your AWS environment.

Because Bedrock is fully managed by AWS, you cannot choose the instance type or customize the container or code. Nor can you deploy new models to Bedrock. If you have a proprietary model or a fine-tuned version of a model that isn't provided, you cannot serve it via Bedrock; read the following sections to learn to host your custom model.

When to use

If you need immediate inference from high-quality foundation models and want the simplest integration possible, a fully managed solution like Bedrock is a great choice. It shines for use cases like building a quick prototype of a chatbot, text generator, or image generator using pretrained models, especially when you don't want to manage any infrastructure.

However, it's not suitable if you need to deploy your own custom model or if you require extensive custom inference logic. In such cases, other serving options provide more flexibility.

Option 2: One-Click Foundation-Model Deployment

Another no-code/low-code model-hosting method is Amazon SageMaker JumpStart. Like AWS Bedrock, SageMaker JumpStart provides a model hub with pretrained models and easy deployment mechanisms. It also gives you a large catalog of models, including AWS-curated foundation models like Cohere, Falcon, Llama, and Stable Diffusion, as well as models from the Hugging Face Hub.

The big difference between JumpStart and Bedrock is that JumpStart lets you deploy these preselected models to your own SageMaker infrastructure with minimal coding. It's often described as *one-click deployment* for models, and it's available through the SageMaker Studio UI as well as programmatically via the SageMaker SDK.

If you're hosting models in your own AWS account, you'll pay by the hour for the SageMaker-managed instances. This means a bit more infrastructure management, since you now host a model endpoint yourself. You can choose the server instance types, such as g5.48xlarge and g6e.48xlarge.

In the example that follows, you can configure a few more things about the infrastructure than you could in Bedrock, such as the number of model serving instances, the logging levels, and the server instance type:

```
# Define the model ID and version
model_id = "huggingface-llm-mistral-7b-instruct"
version = "*"  # Use the latest version

# Create a JumpStartModel instance with hardware configurations
model = JumpStartModel(
   model_id=model_id,
   model_version=version,
   instance_type="ml.g5.2xlarge",  # Specify the instance type
   role=sagemaker_execution_role, # Specify the execution role
   env={ # Example environment variable
       "SAGEMAKER_MODEL_SERVER_WORKERS": "1",
       "SAGEMAKER_CONTAINER_LOG_LEVEL": "20"
   },
   # Add other configurations as needed.
)

# Deploy the JumpStart model
predictor = model.deploy(
   initial_instance_count=1,
   endpoint_name="my-mistral-endpoint",
   role=sagemaker_execution_role,
   sagemaker_session=sess
)

# Send prediction request
response = predictor.predict({"inputs": "Hello, world!"})
```

Limitations

Under the hood, SageMaker JumpStart uses the standard SageMaker inference infrastructure but automates much of the setup. It provisions the server instances, downloads model artifacts and Docker images, and hosts the model as a model endpoint. Once deployed, the endpoint works like any other SageMaker endpoint: you pay for the uptime of the instance.

JumpStart is easy to use, but comes with some limitations on customization, Since it automates nearly every step of model deployment, you have limited knobs you can tune to customize your model. For example:

- Not every model in the catalog supports fine-tuning or evaluation; some are deploy-only.
- Instance choices are constrained per model; models have their own supported instance types.

- You cannot customize the default model inference behavior, such as data preprocess and batching.
- You can't adjust prediction requests' payload schemas or content types.
- You can't choose your CUDA or PyTorch version.
- It's difficult to apply optimization techniques since JumpStart configures serving options, such as context length and batch size, for the selected model.

If you need complete control over the inference logic, or if you need to serve a model that JumpStart doesn't support, continue on to learn about more customizable SageMaker options.

When to use

If you want to get a popular pretrained model up and running in your own AWS environment quickly, JumpStart is ideal.

JumpStart targets no-code or low-code users who want to avoid dealing with container setup or writing inference code for well-known models. For instance, if you need a BERT model for text classification or Stable Diffusion for image generation and don't want to manage the serving stack, JumpStart can deploy it for you. It's also useful as a learning tool for SageMaker, since it sets up endpoints following best practices.

Option 3: Bring Your Own Model

For a still more customizable option to serve your own model or models that are not listed in JumpStart's or Bedrock's model catalogs, AWS SageMaker provides a variety of prebuilt serving Docker images called Deep Learning Containers (DLCs) that you can use to deploy models without writing custom inference code.

DLCs include framework-specific containers such as TensorFlow, PyTorch, and Hugging Face Transformers, as well as SageMaker's built-in algorithm containers, such as the DJL container. Think of this as a "bring-your-own-model" approach, but *not* "bring-your-own-code."

With DLC, you can serve a trained model artifact (perhaps one you trained yourself or obtained elsewhere) without worrying about model serving code, since the DLC knows how to handle that model type. You can use this for custom models (such as your own PyTorch model) or open source foundation models (for example, deploying a model from Hugging Face Hub) with minimal coding work.

The key difference from JumpStart is that, here, you are explicitly *choosing* which container (serving framework) to use and managing the deployment steps yourself, though still in just a few lines of code. JumpStart would autoselect those for you. This gives you a bit more flexibility—for example, you can specify exactly which version of

PyTorch or Transformers library you want, or deploy a model that isn't in the curated JumpStart list. However, to use this option, you need to understand the container's requirements.

Let's walk through an example of hosting a PyTorch model with TorchServe. The first step is to select a prebuilt DLC that matches your framework and Python version. You can use the following code to search for a suitable DLC image for your PyTorch model:

```
# Find the available serving image by searching model framework
# and server instance type
baseimage = sagemaker.image_uris.retrieve(
        framework="pytorch",
        region="",
        py_version="py310",
        image_scope="inference",
        version="2.0.1",
        instance_type="ml.g4dn.16xlarge",
    )
```

Next, create a serving image and a `Model` object with model file location in AWS S3. Deploy the model object to the selected instance type (for instance, g4dn.16xlarge). The `model.deploy` function will handle resource provision and service deployment on AWS:

```
# Create the Model object
model = Model(model_data = f'{output_path}/mnist.tar.gz',
                  image_uri = baseimage,
                  predictor_cls = Predictor,
                  name = "mnist")

# Deploy model
endpoint_name = 'torchserve-endpoint-1'
predictor = model.deploy(
    instance_type='ml.g4dn.16xlarge',
    initial_instance_count=1,
    endpoint_name = endpoint_name,
    serializer=JSONSerializer(),
    deserializer=JSONDeserializer())
```

Here's another example: serving an open source LLM (Llama) from Hugging Face using AWS's prebuilt Large Model Inference (LMI) containers, powered by the DJL serving framework with vLLM:

```
# Create the SageMaker Model object.
# In this example we let LMI configure the deployment settings
# based on the model architecture
model = DJLModel(
  model_id="meta-llama/Meta-Llama-3.1-8B-Instruct",
  env={
    "HF_TOKEN": "",
```

```
        # Add more serving configurations here
        # Example: set tensor parallel degree
        "OPTION_TENSOR_PARALLEL_DEGREE": "4",
        # Example: specify serving loader as vLLM
        "OPTION_SERVING_LOADER": "vllm",
        # Example: set max rolling batch size
        "OPTION_MAX_ROLLING_BATCH_SIZE": "128",
    }
)

# Deploy your model to a SageMaker Endpoint
# and create a Predictor to make inference requests
endpoint_name = sagemaker.utils.name_from_base("llama-8b-endpoint")
predictor = model.deploy(
    instance_type="ml.g5.12xlarge",
    initial_instance_count=1,
    endpoint_name=endpoint_name)
```

This approach requires a more explicit setup than JumpStart does: you need to search for and specify the appropriate DLC container and version to match your model's framework, and ensure your model artifact is properly prepared. In return, you gain flexibility—for example, you can use newer releases of frameworks not available through JumpStart's simplified interface. You still don't need to write serving code yourself, since the container provides the implementation.

Limitations

The main limitation of the DLC approach is that you have little control over the container internals. Frameworks, NVIDIA drivers, CUDA, Python, and the base OS are all pinned to the image tag (for example, TensorFlow 2.19 with CUDA 12.2 on Ubuntu 22.04). To mix versions of the internal libraries, you must build your own image.

SageMaker also enforces a fixed port and HTTP contract: each DLC container must expose a web server on port 8080 and implement `/invocations (POST)` and `/ping (GET)`. These cannot be changed without leaving the standard path, and default response timeouts apply (for example, 60 seconds for standard responses).

Another constraint is that DLCs assume specific input/output formats for prediction requests. If your use case requires custom preprocessing or postprocessing, you'll need to either provide a custom inference script (Option 4) or handle those steps outside the model.

When to use

This prebuilt serving container (DLC) approach is ideal when you have a model trained with a common framework, you want to deploy it quickly using SageMaker's managed service, and you might need slightly customized choices. For example:

- If you've fine-tuned a Hugging Face Transformers model (like BERT or Qwen3) and have the weights, you can use the Hugging Face inference DLC to serve it with minimal effort.
- If you want to deploy a TensorFlow SavedModel or a PyTorch *model.pth* file, you can use images like TorchServe or TensorFlow Model Serving to serve it with minimal effort.

To use the DLC prebuilt serving images approach, your model must be in a supported format and you must be OK with the container's default prediction handling. If you need to preprocess input or postprocess output in non-AWS-standard ways, you might eventually require the next level (custom inference script, Option 4). But if the default serving logic is sufficient, this saves you from writing any inference code.

Option 4: Bring Your Own Code

As we move up the customization ladder, the next step is to write your own serving code. In AWS SageMaker, this is done through Script Mode for inference. With Script Mode, you still rely on a SageMaker-provided container for the underlying framework, but you supply an entry-point script that implements the logic for model loading and prediction. This gives you much greater flexibility: you can define custom preprocessing and postprocessing, support nonstandard input/output formats, and even load multiple models within a single container. The trade-off is that you now need to write and maintain code, whereas the earlier options required no custom inference code.

As an example, let's use a SageMaker Large Model Inference LMI container to host an LLM with our own serving implementation. For brevity, we'll illustrate this concept with pseudocode.

The first step is to define the per-model serving configuration. The following code specifies both the serving engine and location of the model:

```
%%writefile my-own-llm/serving.properties
engine=Python
option.tensor_parallel_degree=2
option.rolling_batch=vllm
option.s3url=s3://sagemaker-us-west-2-/large-model-lmi/code/my-own-llm
```

Next, we implement our own model serving (loading and execution) code in a file called *model.py* and implement the model serving execution entry point at the `handle` function:

```
%%writefile my-own-llm/model.py
import os
# ...
PAD_TOKEN_ID = 50256
```

```
# initialize model
def initialize(properties):

    model = AutoModel.from_pretrained(model_id)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    model.eval()
    .. ..

# execute model
def run_inference(input_texts, onnx_model, tokenizer):

    max_batch_size = 128
    z = torch.empty([0,768]).to("cuda")
    for i in range(0, len(input_texts), max_batch_size):
        logging.info(f"Start Iteration: {i}")

        start_time = time.time()
        batch_dict = tokenizer(input_texts[i:i+max_batch_size],\
             max_length=512, padding=True, truncation=True, \
             return_tensors='pt').to("cuda")
        logging.info(f"After Tokenize Latency: {time.time() - start_time}")

        start_time = time.time()
        with torch.no_grad():
            outputs = model(**batch_dict)

        # ... ..
        z = torch.cat((embeddings,z), 0)
    results = [{"embedding": embedding.tolist(), "index": idx} \
        for idx, embedding in enumerate(z)]
    return {"embeddings": results}

# model execution entry point
def handle(inputs: Input) -> None:
    logging.info("handle : Handle start...")
    global model, tokenizer
    if not model:
        logging.info("handle : initializing model")
        model,tokenizer = initialize(inputs.get_properties())

    if inputs.is_empty():
        logging.info("handle : inputs is empty")
        # Model server makes an empty call to warmup the model on startup
        return None

    # implement model request pre-process logic
    data = inputs.get_as_json()
    input_sentences = data["inputs"]
    # run model inference
    res = run_inference(input_sentences, model, tokenizer)
```

```
    logging.info("handle : Handle End")
    return Output().add_as_json(res)
```

Next, we package *mode.py*, the model weights, *serving.properties*, and other model configuration files into one file and upload it to S3 cloud storage:

```
# package model file
%%sh
tar czvf my-own-llm.tar.gz my-own-llm/

# upload model file to S3
s3_code_prefix = "large-model-lmi/code/my-own-llm" # increment the version
bucket = sess.default_bucket()  # bucket to house artifacts
code_artifact = sess.upload_data("my-own-llm.tar.gz", bucket, s3_code_prefix)
```

Once we have the model file with the customized serving code and configuration, the last step is deployment, which is very similar to Option 3, including choosing a serving image, creating a model object, and deploying the model:

```
# choose an LMI image.
image_uri = image_uris.retrieve(
    framework="djl-deepspeed",
    region=sess.boto_session.region_name,
    version="0.25.0"
)

# create model object
model = Model(image_uri=image_uri, model_data=code_artifact, role=role)

# deploy model to an endpoint
predictor = model.deploy(initial_instance_count=1,
             instance_type="ml.g5.2xlarge",
             endpoint_name="my-own-llm-128")
```

Limitations

In many cases, the "Bring Your Own Code" approach strikes a balance: you can tailor the inference logic to your specific model and use case while reusing the cloud vendor's serving stack (AWS LMI or DLC containers) and infrastructure (SageMaker model deployment).

The trade-off lies in what you inherit from the vendor's container. You're limited to the frameworks, Python/OS/CUDA versions, and serving libraries baked into that image, and you cannot change the HTTP interface or request schema. If you need full control—such as using a preferred library like vLLM, customizing the runtime stack, or defining your own APIs—you'll need to move to Option 5: Bring Your Own Serving Image.

When to use

Essentially, when your prebuilt containers' limitations (see Option 3) become a bottleneck, Script Mode lets you override the per-model serving logic in the DLC without building a container from scratch.

More specifically, use this option when you have a model that can run in one of the supported SageMaker containers, but the inference process needs custom steps or is not covered by the default container behavior: for example, if you need to transform the input data significantly before feeding it to the model (such as custom encoding and image transformations) but process the model output similarly (like filtering or formatting results).

Option 5: Bring Your Own Serving Image

At the top end of our first five options for customization and control, the most customized option is to bring your own Docker container to SageMaker. In this scenario, you are responsible for providing a container image that can serve your model.

SageMaker will still manage the deployment of this container to a cluster of instances and provide the endpoint URL, but everything inside the container is up to you. This is the most flexible option: you can use any programming language, any framework (even ones not directly supported by SageMaker's built-in containers), and any custom code for inference. You also can install any system dependencies, use custom network configurations, and so forth within the container.

Essentially, SageMaker treats your container as a black box that adheres to a simple contract: HTTP requests come in, responses come out on certain ports and paths. This is where the single-model and multi-model serving services we introduced in Chapter 3 fit in: you can put the service serving code into the Docker container and still use SageMaker to deploy and host the service and models.

Limitations

The main drawback of bringing your own serving image is the added complexity: you must build, maintain, and update the container yourself, while ensuring it complies with SageMaker's requirements, such as its health check and invocation endpoints. Testing and debugging are also more involved—you'll need to write a Dockerfile, implement the app, push the image to AWS Elastic Container Registry (ECR), and work with more verbose API calls. This overhead is the trade-off for having full control.

Another limitation is in distributed serving, where performance depends on coordination across serving instances. If you want to implement optimizations such as

distributed KV caching, prompt caching, or request routing, it becomes necessary to move to Option 6.

When to use

In short, whenever the constraints of SageMaker's provided containers become too limiting, "Bring Your Own Serving Image" is the escape hatch. With this option, you can customize virtually anything inside the container—for example, you could adopt a different ML framework or implement performance-critical logic in C++. As long as your container complies with SageMaker's required HTTP interface, it will run.

Some common scenarios include:

- Your model depends on a stack or serving framework that SageMaker doesn't support, isn't yet available in the latest version, or has been heavily customized.
- You need complete control of the inference environment, beyond model code—such as system-level optimizations or running on a specific Linux distribution.
- You want to bundle multiple processes or services in the same container for advanced use cases, like running an auxiliary service alongside the model to collect specialized metrics based on price or user metadata.

Option 6: Build Your Own Serving Infrastructure

The first five options all leverage parts of AWS's serving stack. But if you want to operate a full-fledged model serving platform—like the one outlined in "LLM Serving in Enterprise Systems: An Overview" on page 122—you can build your own services directly on top of cloud infrastructure. For example, you could use a cloud-managed Kubernetes service.

In this model, you build custom serving images, choose your runtime (such as Triton, vLLM, TensorRT-LLM, KServe, or Ray Serve), and run them on GPU node groups. You also take ownership of traffic management, autoscaling, security, observability, and cost controls. At the same time, you can still lean on AWS primitives and managed components, such as EKS, ALB/NLB, ECR, S3, EFS/EBS, IAM/IRSA, Karpenter/Cluster Autoscaler, and CloudWatch, and you can use Kubernetes add-ons like Prometheus/Grafana, OpenTelemetry, Fluent Bit, CNI/NetworkPolicy, Gatekeeper/Kyverno, and Argo CD/Rollouts.

Use this model when:

- You require full control of the serving runtime, including kernels, batching, tokenization, custom sidecars, or nonstandard APIs such as gRPC or Server-Sent Events (SSE).

- You need aggressive cost and performance tuning, such as spot GPU utilization, GPU sharing (MIG/MPS), model bin-packing, or custom autoscaling based on tokens per second (TPS).
- You must meet strict compliance or data isolation requirements, including private clusters, VPC-only egress, per-tenant isolation, or custom audit trails.
- You want to run on cutting-edge hardware or implement specialized optimizations not yet supported in managed services—for example, speculative decoding, KV-cache sharding, or advanced routing and load-balancing strategies.

Comparing the Options

There is no one-size-fits-all serving solution. Each company and each team has a different capacity, business model, and project timeline. When we evaluate which option will work best in a specific case, we normally start from the trade-off between ease of use and control, then evaluate from several other perspectives, such as cost and performance.

Figure 4-8 presents a decision tree to guide you in choosing among the available options, beginning with ease of use.

By following this decision tree, you can make serving choices based on ease of use—that is, how little development work and customization are required. The more AWS services and components you leverage from, the faster you can get a model up and running.

In practice, however, development speed is only one factor. Operational and maintenance costs are equally important. Many teams start with the simplest option to validate an idea, then shift toward more customized solutions as projects mature or requirements grow.

For example, you might initially run the Qwen3 model on Bedrock, which charges at a rate of $0.10 per input million tokens. As usage (throughput) increases, you might decide to move to option 2 or 3 to host Qwen3 in your own account, paying $1.172 per hour for server instances, a reduction compared to the per-token cost. Later, as business needs demand higher throughput and lower latency, you could transition to options 4, 5, or 6, depending on the level of customization required.

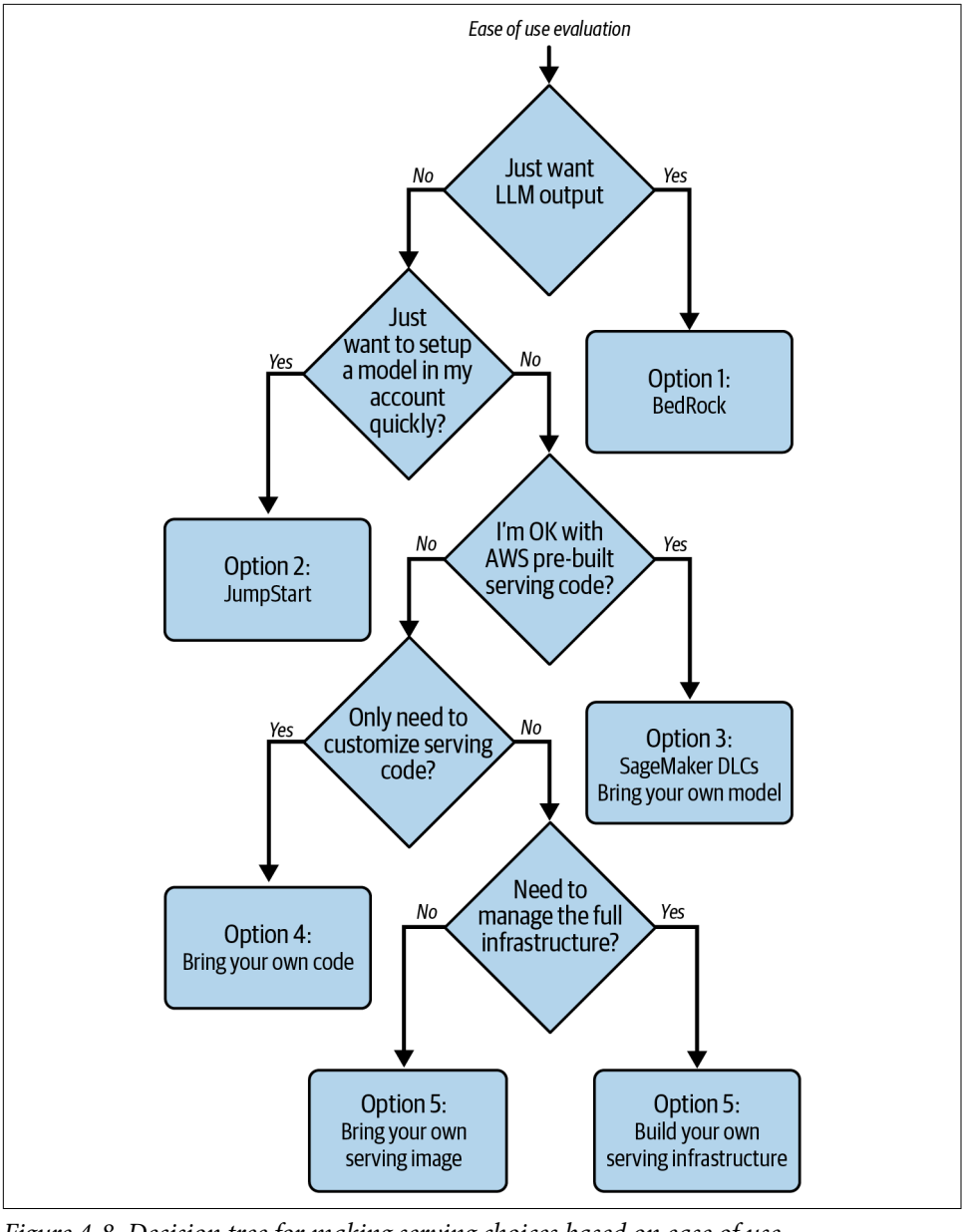

Figure 4-8. Decision tree for making serving choices based on ease of use

Build or Buy? Understanding Strategies

We've heard lots of people debate whether to build their own serving stack or use services provided by cloud vendors. Both authors of this book have been building and maintaining different model serving systems for more than a decade, and we can tell you that this is *not* binary. The choice between built and rent is a *spectrum*, *not a switch*!

The better you understand how to build a serving stack from first principles, the better you can evaluate managed options from cloud vendors, spot hidden trade-offs, and decide *how much* to customize at any moment. As with the decision tree from Figure 4-8, think in terms of degrees of control, rather than either/or.

In practice, most teams operate somewhere in the middle: they run on vendor platforms while adding targeted customizations such as custom handlers, autoscaling signals, routing logic, or cost controls. A full "build your own" (BYO) approach is usually reserved for cases where control, large-scale cost efficiency, or strict compliance requirements make it necessary. For example, you might choose BYO to avoid vendor lock-in and ensure your system can run across multiple cloud providers.

Why Knowing How to Build Helps—Even If You Won't Build

If you're using a cloud vendor, even though you won't need to build everything yourself, understanding a homegrown serving stack (as described in Chapters 2 and 3) is critical. This understanding lets you:

Decode vendor features
: You'll recognize which knobs map to batching, caching, quantization, and adapter loading, and where the ceilings are.

Quantify trade-offs
: You'll be able to perform a good trade-off analysis between different vendor features and approaches, such as "bring your own model" versus "bring your own code."

Customization
: You can keep the vendor defaults for 80% of paths and surgically replace the 20% that need deeper control.

Debugging and troubleshooting
: You can get things working or fixed quickly in the vendor system. Vendor solutions often work like black boxes, with limited documentation for their configuration and internal logic. Logging can be another pain point, since not all the logs are captured and displayed.

Our Selection Strategy

Here is an approach we've found very useful in practice, when we discuss with clients whether to stay with their vendor-provided serving solution or move to a more customized solution:

- Stay vendor managed if your service-level objectives (SLOs) (such as prediction latency and throughput) are met, costs are acceptable, and feature velocity and timelines matter most.
- Hybridize when a few model endpoints need special batching, routing, or performance tuning or per-tenant isolation.
- Go BYO when you need control over hardware, runtime, or sovereign networking, or when the cost of serving demands deep tuning.
- Go back to the vendor-managed approach if the volume stabilizes at a low rate or if the system's complexity isn't paying off.

When you learn to build a serving system, you can choose how much to build and customize at any point. The winning strategy is a *dynamic* one: keep a stable API and shared telemetry, customize where it pays off, and expect your position on the spectrum to change as your product, traffic, and constraints evolve.

Measuring Performance in LLM Serving

Throughout this chapter, we have discussed agentic workflows, layered enterprise architectures, and build-versus-cloud decisions. While these design choices shape how an LLM serving system is structured, they ultimately must be evaluated through measurable outcomes. In production systems, architectural elegance alone is not sufficient—performance determines feasibility, user experience, and cost-to-serve.

This is why performance metrics are not an afterthought, but a guiding principle.

Agentic systems amplify latency across multistep reasoning chains. Enterprise platforms must balance scalability with cost efficiency. Build-versus-cloud decisions often hinge on whether a system can meet SLOs within budget. In all these cases, we need quantitative measurements to determine whether a serving design is working as intended.

To prepare for the LLM optimization techniques we'll show you in the next two chapters, we now introduce two critical metrics we have consistently relied on while operating model serving systems in production. These measurements allow us to evaluate architectural trade-offs, benchmark improvements, and quantify the impact of optimization techniques. They are:

Latency
: The time required to process and generate responses

Throughput
: The number of requests or tokens processed per unit of time

Latency Metrics

When measuring the speed of LLM prediction execution, the three most important metrics are end-to-end latency, time to first token (TTFT), and time per output token (TPOT). Let's look at them one by one:

End-to-end (E2E) latency
: *E2E latency* is a relatively general term used to measure the total computation time taken for both ML and non-ML workloads. In the context of LLMs, it refers to the time from when the model receives a request to when it completes generating the full response. E2E latency can also include the overall serving latency when needed, which accounts for not only model execution time but also additional delays (processing time) in the model serving system, including request queuing (when taking big loads of requests), network delays, routing time, scaling overhead, and other various system-level factors.

Time to first token (TTFT)
: *TTFT* refers to the time elapsed between the model receiving a request and emitting the first token. From users' perspective, in chatbot-like use cases, it translates to how *responsive* the LLM model is—meaning how quickly the customer will see the first output token returned from the model. It also corresponds to the prefill phase we discussed in Chapter 2, where the model processes the input prompt, attends to its full context, understands it, and prepares to generate output. In practice, TTFT is mainly used when discussing model latency.

Inter-token latency (ITL) or time per output token (TPOT)
: *ITL* (or *TPOT*) measures the time taken to generate all subsequent tokens after the first. It corresponds to the decode phase (see Chapter 2), where the model generates tokens sequentially, typically at a relatively steady rate. This metric is crucial for understanding the efficiency of an LLM's autoregressive token generation. Like TTFT, ITL/TPOT is also mostly relevant when discussing model latency.

Figure 4-9 shows how TTFT and ITL map to the prefill and decode phases within overall end-to-end latency in an LLM execution. The time costs of tokenization and detokenization are also captured in the graph.

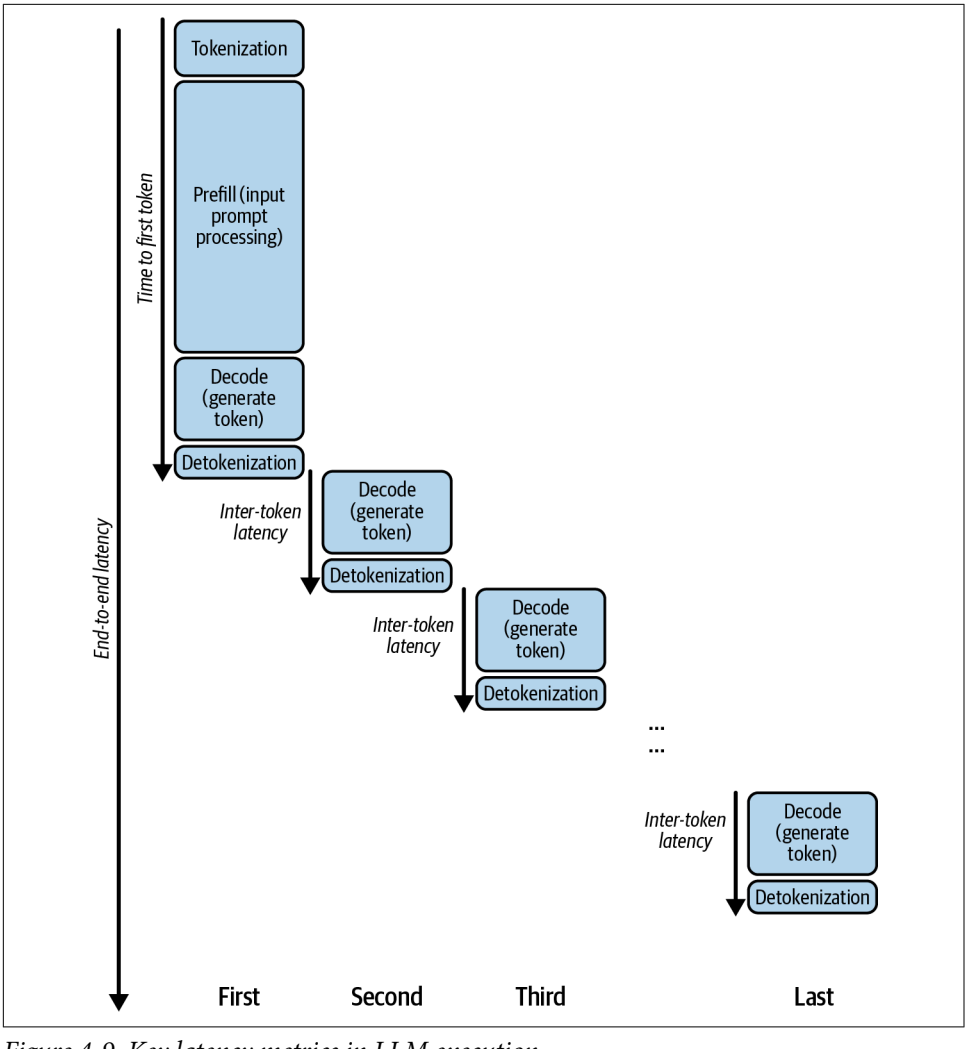

Figure 4-9. Key latency metrics in LLM execution

In practice, tokenization and detokenization times are typically minimal compared to model inference time, and are often ignored from TTFT and ITL calculations.

Now that you're clear on the latency-related concepts, let's look at the formulas to calculate them. We'll assume that ITL is consistent; call the total number of tokens generated from the LLM *N;* and ignore tokenization and detokenization time:

The formula for E2E latency is:

$$\text{E2E latency} = \text{TTFT} + \text{ITL} \times (N - 1)$$

To calculate TTFT, the formula is:

> TTFT = full tokenization of the input + prefill + decode first token + detokenize first token = prefill + decode first token

Finally, to calculate ITL:

> ITL = decode one new token + detokenize the new token – detokenize the last token = decode one new token

In real-world use cases, all three latency metrics—E2E latency, TTFT, and ITL—are important, but their priority varies depending on the application and use case.

For example, if an LLM call is part of an agentic workflow where the downstream process depends on the full model output, then minimizing E2E latency becomes the primary objective. On the other hand, in chatbot applications, where responses are streamed back to users in real time, TTFT can have the most significant impact on user experience because it directly corresponds to the model's responsiveness. But if the output is very long, ITL becomes a critical factor as well, since a very high ITL can make the response feel sluggish.

In the following chapters, we will explore techniques for improving LLM serving latency. However, not all optimization techniques will enhance all three metrics at the same time. Certain trade-offs need to be made, so selecting the right techniques will depend on the specific requirements of the use case.

Throughput Metrics

Throughput is one of the core metrics in LLM serving. It's used to answer the question: "How many predictions can the system handle over time?" This tells you how efficiently your serving system uses resources to deliver results at scale. The commonly used throughput metrics are:

Requests per second (RPS) or requests per minute (RPM)
: RPS or RPM are universal throughput metrics for both ML and non-ML workloads. These metrics quantify how many requests the model or the entire serving infrastructure can handle within a given timeframe. But to LLM serving, RPS and RPM have quite a lot of drawbacks and lack of granularity specific to LLM serving, which we will discuss later.

Tokens per second (TPS)
: TPS is a throughput metric specifically used for LLM generation. It measures the number of output tokens the model or the LLM serving service generates per second. One thing to keep in mind is that TPS only counts *generated* tokens: not input tokens or the sum of input and output tokens.

Both RPS/RPM and TPS have advantages and limitations, and neither is a perfect standalone measure of efficiency in LLM serving. Let's look at why.

First, RPS and RPM are sensitive to prediction request patterns, since long inputs and outputs can negatively affect LLM serving performance if measured per request. As a result, RPS and RPM heavily depend on factors like input and output length, as well as the number of concurrent users. This means direct RPS/RPM comparisons between two different workloads with different traffic patterns are basically useless: it's not an apples-to-apples comparison.

Second, TPS is useful, but it has risks because it can be manipulated. TPS is a standardized metric in the LLM space, and in a lot of cases we also leverage it to calculate cost per token. However, it can be artificially inflated in a few ways:

- Reducing input length significantly lowers TTFT, which in turn reduces the per-request workload. This makes the TPS appear higher than it should.
- Larger or optimized batch requests can also increase TPS. Increasing batch sizes to maximize model process parallelism or making the batched request have uniform input and/or output lengths can improve TPS, since doing so improves GPU utilization and reduces GPU idle time. We will explain why in the next two chapters.

Best Practices for Performance Measurement

When introducing performance measurement, we noted that no single metric is sufficient on its own. Latency and throughput are the two core metrics, but their value depends on how they're interpreted and applied in context. In our experience, the most effective performance optimization comes from combining these metrics with a set of practical best practices. Let's walk through some that we've found especially useful.

Identify the trade-off between latency and throughput

A good starting point is to recognize the trade-off between latency and throughput. Improving one often comes at the expense of the other. For example, for offline, noninteractive workloads such as batch processing, throughput usually matters most because it lowers cost. In contrast, interactive applications like chatbots or real-time agents demand responsiveness, making latency the critical metric.

Set acceptable latency goals based on specific use cases

The next important thing is defining what "good enough" means for your application. Setting clear latency targets prevents unnecessary optimization. For example, in an interactive chatbot, if you've already achieved a 1-second response time, squeezing it

down to 0.5 seconds may not significantly improve user experience. At that point, it's better to shift focus toward optimizing throughput and cost.

Decompose E2E latency into TTFT and ITL

Breaking down end-to-end latency into more specific components—TTFT and ITL—provides a sharper view of performance bottlenecks. These sub-metrics can guide targeted improvements. For instance, in systems with capped output length, TTFT may be more critical than ITL, since users care more about how quickly the first token appears than how quickly the rest are generated.

Simulate real-world traffic patterns as closely as possible

Another key practice is simulating traffic patterns that resemble actual usage. Input and output lengths vary widely across users, and both heavily influence performance. When creating test suites, we need a clear understanding of the use case and its real-world traffic patterns to ensure performance optimizations are meaningful and do not deviate from real usage.

For example, long prompts with short answers behave very differently from short prompts with long outputs. If the setup matches the real usage, you will be optimizing in the correct direction as a whole and making the right decisions and trade-offs.

When simulating traffic, please remember that real-world traffic to systems is never perfectly uniform. Traffic levels can be different throughout the day or at specific times of the year. Bursts of traffic can overload the system, leading to queuing delays or even request failures. So please analyze your traffic pattern carefully before simulating it.

Maintain consistency in your experiments

Consistency across experiments is just as critical. Performance tuning often involves adjusting multiple parameters, and their interactions can be complex. Changing too many at once makes it impossible to isolate effects. What we found useful is to turn one knob at a time to isolate the impact of each optimization. Keeping request patterns and configurations stable across tests ensures reliable, apples-to-apples comparisons.

Monitoring and tracking hardware utilization

Monitoring hardware utilization provides another valuable perspective. Tracking GPU and CPU usage, memory consumption, and related metrics helps identify whether bottlenecks are caused by model behavior or hardware limits. This understanding is key to applying optimizations effectively, as later chapters will show.

Avoid manipulating or inflating performance metrics

Ensuring a fair and representative benchmark is important to prevent you from drawing misleading conclusions. As discussed in "Throughput Metrics" on page 152, performance metrics can be artificially inflated and doing so could backfire later on when the system goes to production or lead the optimization effort in the wrong direction.

Continuously monitor performance in production

Once the model gets deployed, continuous monitoring ensures stable performance and detects possible shifts in usage patterns. Beyond latency, throughput, and hardware utilization, user behavior can also shift over time, causing request spikes and failures.

Run test suites periodically

Even after the model goes to production, regular testing is needed to prevent regressions, drive continuous improvements, and keep the system efficient, scalable, and aligned with evolving workloads.

For example, regression checks can ensure that new updates don't degrade latency or throughput. Scalability tests can simulate peak loads on days like Black Friday or scale down for nonpeak hours. A/B testing can create data-driven, apples-to-apples comparisons on new optimization techniques you want to introduce before you roll them out to production.

In short, meaningful performance measurement requires a mix of granularity, realism, fairness, and ongoing vigilance. By applying these best practices, you'll avoid misleading results and instead gain insights that translate into real-world improvements in latency, throughput, and cost efficiency.

Summary

In this chapter, we built on Chapters 2 and 3 (inference internals and single/multi-model design) and stepped you up to production best practices for LLM serving—agentic use, enterprise architecture, build-versus-cloud choices, and the metrics you'll use to optimize serving performance.

We began with model serving in agentic applications, walking through a minimal Knowledge Agent that uses OpenAI models, a planner, and actions (RAG for retrieval, summarization, and analysis) to show how serving enables autonomy. We then compared RAG and CAG (preloading knowledge into the context or KV cache) and analyzed their trade-offs.

Next, we presented a layered enterprise architecture whose components can evolve independently, plus an open source Kubernetes stack to implement it. We also covered six AWS serving options spanning increasing customization and provided a decision tree for choosing among them.

A key theme is that the decision between building and using a cloud vendor is a spectrum, not a binary. Most teams operate in the middle—leveraging vendor platforms with targeted customizations—and shift that balance over time based on ease of development, cost, and usage patterns.

Finally, we standardized key metrics (latency and throughput) and best practices for fair benchmarking, realistic traffic simulation, hardware-utilization tracking, and regression/scalability testing. The next chapters will show you how to turn these metrics into concrete latency, throughput, and cost improvements.

CHAPTER 5
Challenges When Serving LLMs

So far in this book, we have demonstrated the core concepts of model serving, provided several architectural patterns for serving ML models, and analyzed the trade-offs involved in deploying models at scale. By now, we hope you have a strong understanding of model serving paradigms, because we are about to take a significant step into a different realm. In this chapter, we will shift our focus to one of the fastest-growing fields in the AI world: optimizing LLMs for serving.

Since the rise of ChatGPT in late 2022, LLMs have transformed how AI is applied in real-world scenarios, from chatbots and code generation to advanced reasoning and decision-making systems. However, their sheer size, computational demands, and unique serving requirements introduce challenges that can go far beyond classic model serving techniques. From novel ideas to widely adopted frameworks, the field of optimizing LLMs for faster and more efficient serving performance has evolved at an unprecedented pace. It can be daunting: anyone not familiar with this area can easily get overwhelmed. For example:

- When reading technical blogs, you may wonder: "What is this vLLM framework that has gained popularity in just a year or two and has already been adopted in so many places?"
- When reading research papers, you may ask: "How does FlashAttention work, and how can I optimize it at the hardware level to speed up LLM inference?"
- When following AI news, you may come across a new term called MLA and ask: "How much more efficiently does the DeepSeek V3 model run than other attention mechanisms in the serving phase?"

In the following chapters, we will introduce you to all these advancements. But before we go into each technique, we need to establish a strong foundational understanding and gradually move you from basic concepts to more complex techniques. This

chapter bridges between all the serving concepts, principles, and paradigms you've seen in the prior chapters and all the advanced LLM optimization techniques we will cover in the following chapters.

You first need to understand the landscape of LLM serving—why it is important, what the hardware requirements are, and how to build your intuition about model optimization. Specifically, we will cover why serving an LLM efficiently can be crucial to the success of your application and business at the very beginning. We'll look at modern hardware's role in LLM serving next, exploring AI accelerators such as GPUs to understand their memory intricacies, compute capability, and interconnect functionality. Then we'll cover:

- The major bottlenecks in LLM serving and how to mitigate them
- Constraints when loading LLMs for serving
- Bottlenecks when executing LLMs, specifically in prefill and decode
- Why model serving can become bottlenecked in different phases

To discuss the last point, we introduce a new concept called *arithmetic intensity*.

Understanding these concepts is essential because, without the fundamentals, optimizing LLM serving can become a tiring trial-and-error experiment. It's easy to get stuck in local optima without knowing it or just knowing how but not why.

By the end of this chapter, you will have the intuition you need to navigate the world of LLM optimization with confidence. The next chapters will build upon this foundation by going through the various techniques needed to improve performance, reduce costs, and deploy LLMs efficiently for the workload your application requires in production in greater detail.

Why Optimizing LLM Serving is Important

The previous chapters have covered how to make ML models functionally operational and well-designed when deploying them to production. Another crucial aspect is ensuring that the ML model runs quickly and efficiently in a production environment. This is especially important for serving LLMs, which require substantial compute power and memory from hardware.

To better understand their impact, we categorize the key factors into three aspects:

- Customer experience
- Cost efficiency
- Scalability, peak load handling, and feasibility

Customer Experience

Customer experience is key to any product's success and can be highly correlated with model response latency. Imagine asking a question in ChatGPT and having to wait over 20 seconds to receive the first token. Such a delay is unsatisfactory, and customers can easily become frustrated while waiting. If we can optimize the serving performance to reduce the model's latency from 20 seconds to just 1 second, while keeping the same hardware setup and overall throughput, that would be a game-changer for the product's success.

However, extremely fast response times may not always yield significant benefits. For example, if we reduce the time to receive the first token from 0.1 seconds to 0.01 seconds, the difference is so small that it doesn't register in our human perception. In this case, if we can trade off some latency for higher throughput, increasing it from 0.01 seconds to 0.1 seconds, we will achieve better cost efficiency.

Figure 5-1 illustrates the relationship between latency versus customer satisfaction. As you can see, the two are inversely correlated—when latency is high, customer satisfaction drops significantly, approaching zero. As latency decreases (moving from right to left along the x-axis), customer satisfaction improves. However, as latency continues to decrease, the incremental gains in customer satisfaction become a lot less pronounced.

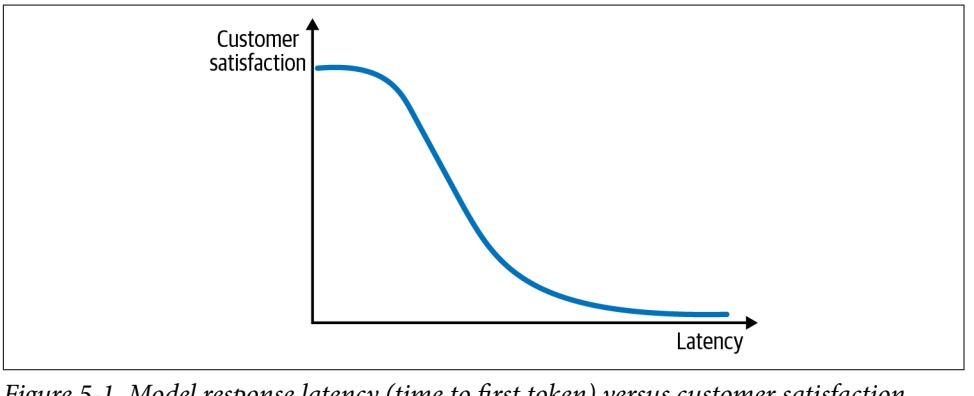

Figure 5-1. Model response latency (time to first token) versus customer satisfaction

Another aspect of customer experience is generation quality. Within the same family or similar families of models, the larger models generally produce higher-quality responses. Figure 5-2 compares the benchmark scores of Llama-3-8B and 70B, demonstrating how larger models outperform smaller models in the same family.

Without optimization, we might be limited to an 8-billion-parameter model to meet latency requirements. However, after extensively optimizing the whole model serving system, we could achieve the same latency and throughput on the same hardware

setup while leveraging a model with 32 billion or even 70 billion parameters. This can significantly improve response quality.

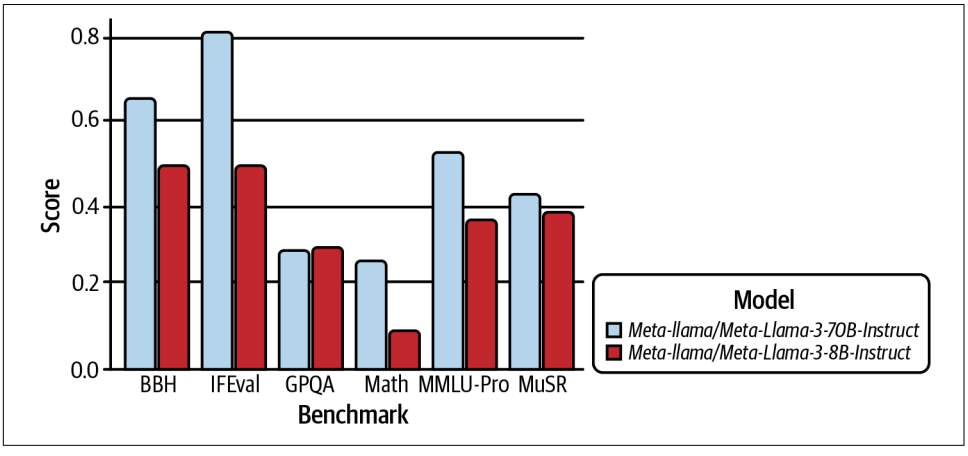

Figure 5-2. Model quality score comparison between Llama-3-70B and Llama-3-8B (adapted from the Open LLM Leaderboard Model Comparator)

Cost Efficiency

Beyond delivering a great customer experience, a product must also be financially viable. No matter how powerful or valuable an AI system is, it cannot succeed if it is too costly to operate at scale. Among all the costs of AI development and operation, model inference is the most expensive component. This might seem surprising, as training often appears to be the dominant cost—especially with reports stating that training costs for LLMs like GPT-4 exceeded $50 million.

However, as shown in Figure 5-3, in which *Signal Integrity Journal* projects AI chip revenue trends from 2024 to 2034, inference hardware consumption already surpasses training today—and the gap is expected to widen even further. This reflects the growing demand for efficient inference infrastructure, as AI applications scale to serve millions of users in real time.

To better understand the dominance of model serving costs in Figure 5-3, let's do some further analysis. Training costs are primarily upfront investments, though some ongoing retraining and fine-tuning costs remain, whereas inference costs scale continuously as usage picks up. Once a model is deployed to production, every single query or interaction incurs ongoing expenses, which accumulate fast as adoption continues to grow.

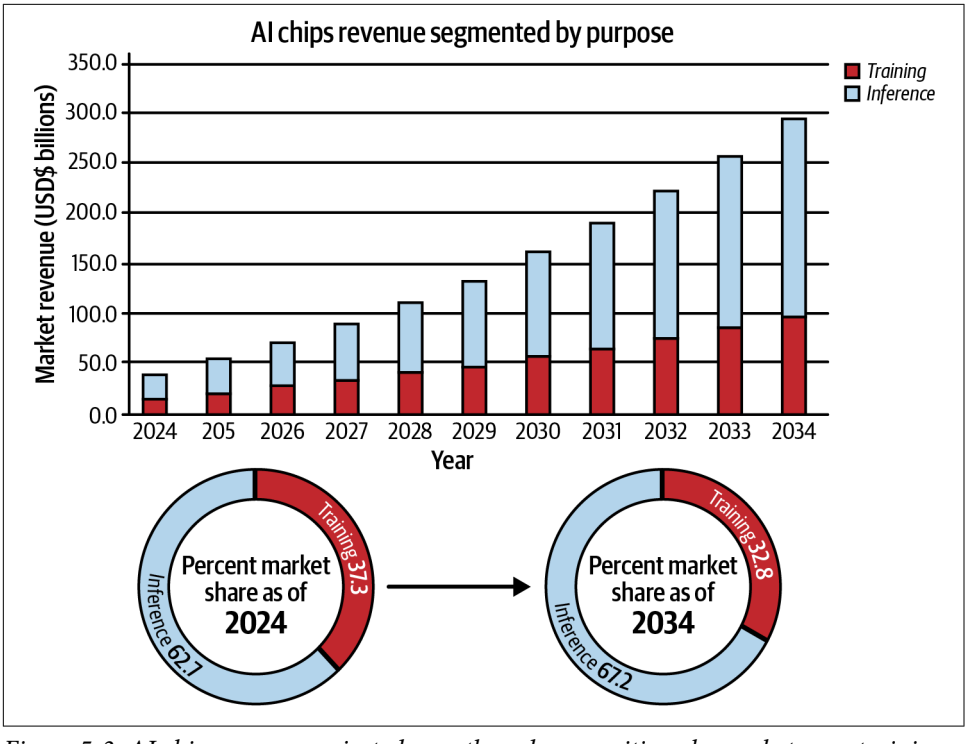

Figure 5-3. AI chip revenue projected growth and composition change between training and inference (adapted from "The Age of Artificial Intelligence: AI Chips to 2034")

Also, at the beginning, a few major players dominated foundation-model training. Recently, however, a growing number of AI labs and open source efforts are also making significant contributions and advancements—though these still account for a small percentage of all the companies using LLMs for business. The vast majority either fine-tune existing models or enhance them with techniques like retrieval-augmented generation (RAG), which can significantly reduce the cost compared to training a foundation model from scratch.

Meanwhile, the demand for inference is nonstop. As GenAI applications proliferate and are applied to more and more industries, the computational burden of LLMs in production is skyrocketing. The rise of AI agents and complex workflows further amplifies this trend, as these systems often require multiple LLMs and embedding model calls within a single workflow. This compounding effect drives inference costs even higher, making efficient model serving a critical challenge for the future of AI.

By optimizing LLM serving, even using the same hardware and achieving similar latency, businesses can potentially achieve more throughput, serving more customers at the same time. This reduces hardware usage when they horizontally scale the serving system, saving a tremendous amount of money! Another example: an optimized model might be able to run on a lower-grade chip instead of a more advanced and expensive chip, while keeping similar latency and throughput. This also saves money.

Scalability, Peak Load Handling, and Feasibility

When a model is deployed in production, GPU demand scales with customer growth. An optimized inference solution enhances customer experience and reduces GPU costs, as you've seen. It also determines whether the system can scale efficiently under heavy traffic, especially when GPU availability is limited—and the industry is indeed just experiencing that.

For example, let's say an LLM-powered sales agent runs smoothly, with stable traffic, most of the year. However, on Black Friday, demand can surge by 400% or more. A less optimized system may really struggle to scale to handle the sudden increase in traffic, leading to bottlenecks, degraded latency, and even request failures. The ability to handle peak traffic reliably is crucial for a resilient and scalable production system. This is a real-world problem we have personally experienced before, and getting the setup right can require a lot of tuning and performance testing to ensure service availability in those periods.

As mentioned in the last section, optimized models can be deployed on lower-grade chips, providing greater flexibility. Even with access to major cloud providers like AWS, Google Cloud, and Azure nowadays, high-end GPUs are not always readily available in every region. The ability to run models efficiently across a broader range of hardware, rather than being constrained to specific high-end GPUs, can be a key enabler for businesses expanding into new markets. We have frequently encountered this issue when deploying applications around the world.

The Role of Accelerator Chips in LLM Serving

Now that you understand the importance of LLM optimization, the next step is to explore the hardware that powers LLMs. Choosing the right accelerator setup is often one of the most important decisions in serving LLMs, because a system's hardware constraints dictate its memory capacity, computational power, and efficiency. Understanding these factors will help you maximize your model's performance and cost-effectiveness.

In this section, we'll focus on NVIDIA's GPU. As we write this in early 2026, NVIDIA's solution for general-purpose computing on graphics processing units (GPGPU) still dominates this market. We will examine the latest trends of AI chips at the end of the chapter, after you gain a good understanding of how models run on those chips.

Let's start by breaking down how to read GPU specifications. GPU specs can be overwhelming due to the sheer amount of technical details, but to simplify things, we'll focus on the attributes most relevant to LLMs: GPU compute power, memory capacity, memory bandwidth, and interconnects. Understanding these factors will help you make informed decisions when selecting a GPU for your LLM inference and training.

After covering those key concepts, we'll analyze and compare the specs of several popular NVIDIA GPUs. We'll also share our insights on which GPUs are best suited for different LLM use cases.

Reading GPU specs

In this section, we will show you how to read GPU specifications by analyzing several attributes of two NVIDIA GPUs: H100 SXM and H100 NVL (Table 5-1). Understanding GPU specifications will not only help you choose the best hardware for AI inference, it will also benefit your model training.

Table 5-1. Specification comparison between H100 SXM and H100 NVL (source: NVIDIA)

	H100 SXM	H100 NVL
FP64	34 teraFLOPS	30 teraFLOPS
FP64 Tensor Core	67 teraFLOPS	60 teraFLOPS
FP32	67 teraFLOPS	60 teraFLOPS
TF32 Tensor Core	989 teraFLOPS	835 teraFLOPS
BFLOAT16 Tensor Core	1979 teraFLOPS	1671 teraFLOPS
FP16 Tensor Core	1979 teraFLOPS	1671 teraFLOPS
FP8 Tensor Core	3958 teraFLOPS	3341 teraFLOPS
INT8 Tensor Core	3958 TOPS	3341 TOPS
GPU Memory	80 GB	94 GB
GPU Memory Bandwidth	3.35 TB/s	3.9 TB/s

GPU compute attribute

GPU compute performance is usually measured in *teraFLOPS*. *Floating-point operations per second* (FLOPS) are a measure of a chip's theoretical GPU compute capability, based on how many floating-point operations (like addition and multiplications) it can perform within a second. In other words, it's a measure of how fast it is, theoretically. *Tera-* means 1 trillion, or 1,000,000,000,000. For example, the top seven

attributes listed in Table 5-1—FP64, FP64 Tensor Core, FP32, FP32 Tensor Core, BFLOAT16 Tensor Core, FP8 Tensor Core, and INT8 Tensor Core—represent the compute performance of different precisions in teraFLOPS.

From the comparison table, you can see that the H100 SXM can perform 1979×10^{12} operations in a second, giving it FP16 precision! (*FP16* stands for "half-precision floating point," more often called *16 bit*. You can double that if you can use a model with FP8 precision. FP16 and FP8 (*8 bit*) are lower-precision numerical formats used in deep learning models. Because FP8 is half the size of FP16, computation can be almost twice as fast. (We will discuss this a lot more in Chapter 6.)

Comparing the FLOPS horizontally between two versions, H100 SXM and H100 NVL, the NVL version is slightly less performant in this aspect. But is it really just worse for all use cases? Maybe not. Let's continue our analysis.

GPU memory attributes

Next, let's look at two equally important, if not more so, GPU specifications: *memory* (also called *VRAM*), which indicates the memory capacity for loading a model, and *memory bandwidth*, which measures the data-transfer speed for GPU computation. H100 NVL has more memory (94 GB) than H100 SXM (80 GB). What's more, it also has a higher GPU memory bandwidth: 3.9 TB/s (terabytes per second), compared to 3.35 TB/s in SXM.

Now we have a choice to make: H100 SXM or H100 NVL? H100 SXM calculates faster than H100 NVL, since it has higher FLOPS, but H100 NVL has better memory capacity and bandwidth. How do we choose between the two?

Let's use an analogy here. Imagine the GPU is a big pizza kitchen, and our goal is to prepare pizzas as quickly as possible to serve as many customers as we can. *Compute power*, measured in FLOPS, represents how fast and powerful our oven is, while the *GPU memory bandwidth* determines how efficiently we can prepare all the ingredients and supply the dough to the oven. *GPU memory capacity* is the fridge's capacity to store all the dough.

Do you see the interaction between all three? If we don't have a big enough fridge to hold the dough, customers will now be sitting there hungry—just as, with a small amount of GPU memory, we cannot load the model. But let's suppose our fridge is just big enough to serve all of our customers. Let's look at the oven.

If we have a very advanced oven, but we can only supply dough very slowly, our powerful oven will be running without baking much pizza. We can't make good use of its power! We'll have a similar problem if we have super high FLOPS for compute, but low GPU memory bandwidth.

Conversely, if the oven isn't very powerful but we can prepare dough very fast, the pizza will take a very long time to bake and customers will have to wait for their food. This is much like having weak compute but high memory bandwidth.

Thus, it's very important to find a balance between compute power, memory size, and bandwidth that fits your workload. We will discuss this more in the next few sections.

GPU interconnect attributes

GPU interconnects enable high-speed data transfer between multiple GPUs, both within a single node and across multiple nodes. It used to be crucial only for model training, but GPU interconnect has become more and more important for model serving.

With the emergence of larger and larger models, one GPU is no longer powerful enough to host most models. You might need to serve a very large model that needs multiple GPUs, with each loading a portion of the model weights or layers. Or when dealing with a latency-sensitive use case, if one GPU is not powerful enough to achieve the latency requirement, you'll need multiple GPUs to work together to achieve faster inference speed. When you face these situations, the GPU interconnect specs become critical. In this section, we will discuss GPU interconnect in detail, starting with *intra-node* interconnects (those within a single node or machine) and then moving to *inter-node* interconnects (those across multiple nodes or machines).

Intra-node interconnects. To help you understand intra-node interconnects, let's look at three variations of the NVIDIA GPU H100 in Table 5-2.

Table 5-2. Comparing GPU interconnects across three types of NVIDIA GPU H100

	H100 PCIe	H100 NVL	H100 SXM
Form factor	PCIe	PCIe	SXM
NVLink support	No (optional)	NVLink Bridge	NVLink/NVSwitch
Interconnect GPU-to-GPU bandwidth	128 GB/s with PCIe	600 GB/s with NVLink Bridge connecting 2 H100 GPUs only	900 GB/s connecting up to 8 H100 GPUs

The first row of Table 5-2 tells you the form factor of the GPU. *Form factor* defines a GPU's physical size, power requirements, and cooling design, which eventually determine how it integrates into a server or workstation. In this example, H100 SXM follows the SXM form factor. SXM-version GPUs are mounted directly onto the motherboard via a custom socket, rather than being inserted into a Peripheral Component Interconnect Express (PCIe) slot. This configuration enables faster connections, better power delivery, and enhanced cooling to achieve better performance with intense multi-GPU workloads.

In contrast, both H100 NVL and H100 PCIe follow the PCIe form factor. PCIe is a commonly used standard data-transfer connection between various electronics. A PCIe version of a GPU means the GPU is installed into a PCIe slot, making it more widely compatible and cheaper, but less performant.

The next two rows show the NVLink technologies each variant uses and the corresponding intra-node GPU-to-GPU bandwidths. *NVLink* is a NVIDIA high-speed interconnect technology that enables direct GPU-to-GPU interconnect for fast, high-bandwidth communication. For the variants without NVLink, GPU-to-GPU communication falls back to PCIe, which is much slower.

Let's look at some GPU interconnect topology diagrams (shown in Figures 5-4–5-7) to understand the difference between NVLink and PCIe, as well as different configurations of NVLink using these three versions of NVIDIA H100 GPUs:

H100 PCIe

In Figure 5-4, two GPUs use PCIe, which has GPU-to-GPU bandwidth of 128 GB/s. (The H100 PCIe can be installed with NVLink Bridge but needs to be bought separately.) This configuration is the cheapest. If you do not need fast GPU-to-GPU communication, such as if you have two small models that run independently on one GPU each, a basic PCIe connection is sufficient.

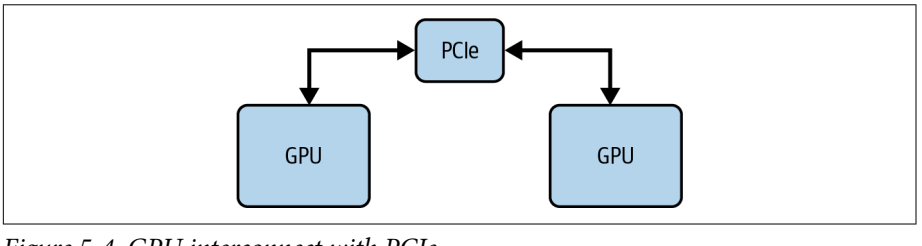

Figure 5-4. GPU interconnect with PCIe

H100 NVL

H100 NVL also follows the PCIe form factor but adds NVLink Bridge. This technology connects a pair of H100 NVL interconnects together with a bandwidth speed of 600 GB/s, as shown in Figure 5-5.

The drawback of the NVLink Bridge setup is that it can only connect two GPUs. If you need more than two GPUs to work together, the slower PCIe connection is still required. For example, in a four-GPU setup, GPU1 and GPU2 are linked with NVLink Bridge, and GPU3 and GPU4 are also linked with NVLink Bridge. This means that communication between GPU1 and GPU2 takes place at 600 GB/s. The same goes for GPU3 and GPU4. But the connections between GPU1 and GPU3, and between GPU2 and GPU4, are still using PCIe, which is 128 GB/s. This setup is a good middle ground that balances cost and performance.

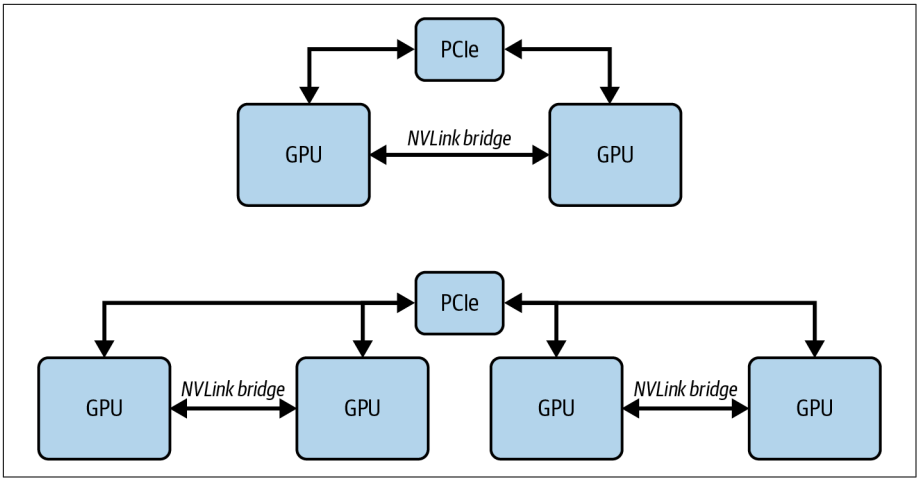

Figure 5-5. Two-GPU (top) and four-GPU (bottom) interconnects with NVLink Bridge

H100 SXM

With the SXM version, up to eight GPUs within the same node can be connected with NVLink, as shown in Figure 5-6.

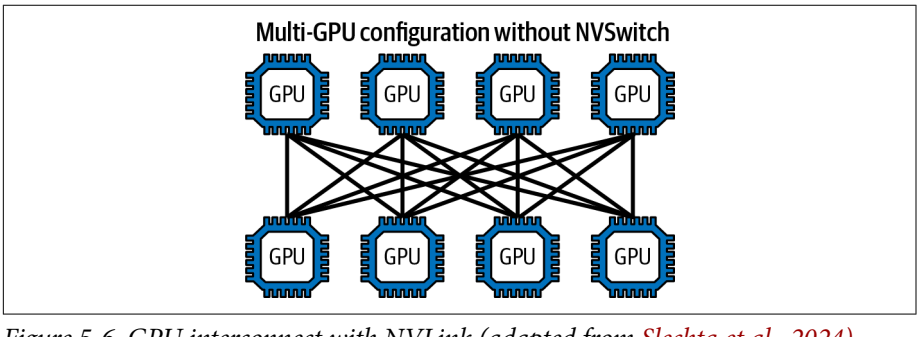

Figure 5-6. GPU interconnect with NVLink (adapted from Slechta et al., 2024)

Figure 5-6 shows how GPUs can be connected with NVLink in SXM. Since SXM can connect eight GPUs and each GPU's interconnect bandwidth is 900 GB/s (see Table 5-2), each GPU must split its 900 GB/s total bandwidth into seven dedicated 128 GB/s (900 GB/s divided by 7) point-to-point connections, each connecting to one of the other GPUs in the system.

If we don't like the fact that we are splitting the interconnect bandwidth by the number of GPUs it's connected to, we can use NVSwitches to achieve full 900 GB/s bandwidth with all GPU connections. NVSwitch is a separate, expensive, high-bandwidth switch that connects multiple NVLink-enabled GPUs together. Figure 5-7 shows that, with four NVSwitches added to the configuration in

between, SXM can achieve full 900 GB/s bandwidth in all-to-all fashion for the best performance possible.

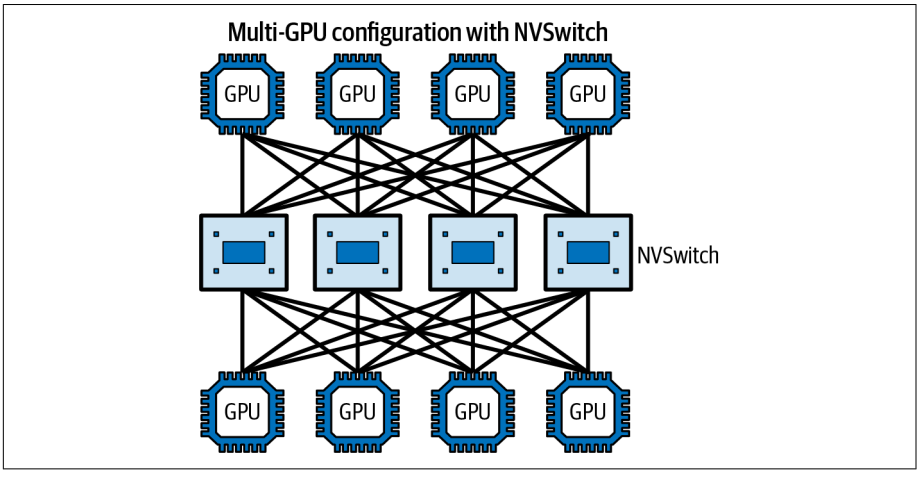

Figure 5-7. GPU interconnects with NVLink and four NVSwitches in between (adapted from Slechta et al., 2024)

Although using an NVSwitch increases interconnect bandwidth greatly, it is the most expensive of all the setups and can be overkill if your workload does not require that much GPU-to-GPU communication.

Inter-node interconnects. All of the setups we just discussed are for intra-node GPU-to-GPU communication. However, it is not possible to have an unlimited number of GPUs in one node. The maximum number of GPUs within a node is usually capped at eight due to physical constraints, power supply, cooling concerns, and software support. As LLMs become larger and larger, eight GPUs in one node can also become insufficient. In this case, you can shard models across multiple nodes, so even more GPUs can work together to serve the model.

Similar to intra-GPU interconnect, inter-node interconnects become important when you're serving models across multiple nodes. One of the best solutions is using InfiniBand (IB) with GPUDirect RDMA. RDMA, which stands for "remote direct memory access," enables direct memory access between GPUs across nodes. NDR 400G InfiniBand, for example, can achieve 50 GB/s across nodes—but that's still quite a lot slower than intra-node performance.

See Table 5-3 for a performance comparison of intra-node versus inter-node bandwidth. For example, within one node, GPU-to-GPU communication can be 900 GB/s with NVLink and NVSwitch while GPU-to-GPU intra-node communication only maxes at 50 GB/s.

Table 5-3. Comparing communication speeds with example H100 setups

Setup	Bandwidth
GPU-to-GPU within node with NVLink/NVSwitch	900 GB/s
GPU-to-GPU within node with NVLink Bridge	600 GB/s
GPU-to-GPU within node with PCIe	128 GB/s
GPU-to-GPU across node	50 GB/s

For LLM serving, the majority of deployments still run on a single node with one to eight GPUs for each model instance (also called a *replica*). As user traffic increases, we scale up the number of model replicas horizontally, by adding more hardware instances with the same setup. The goal of this approach is to minimize unnecessary inter-node communication.

In Chapter 7, we will explore how models can be distributed across multiple GPUs and multiple nodes and cover the pros and cons of different setups and techniques, such as tensor parallelism and pipeline parallelism.

Recent research is exploring separating the prefill and decode phases across different GPUs. Additionally, for large models using a mixture-of-experts (MoE) architecture—like DeepSeek V3/R1—techniques like expert parallelism and data parallelism are applied even across nodes. We will introduce these advanced topics after covering the foundational concepts.

GPU power consumption

One last important constraint is the power consumption of the GPU chip itself, which acts as a hidden but fundamental GPU metric. GPU power is typically measured in watts (W) and is most commonly expressed as *thermal design power* (TDP), which represents the maximum sustained power the GPU is designed to consume under continuous load. Modern datacenter GPUs span a wide range of power envelopes, from a few hundred watts to 700 or more per accelerator, with higher power budgets enabling greater compute density and memory bandwidth. However, this also results in stricter requirements on cooling, power delivery, and system integration.

The importance of GPU power consumption depends strongly on the deployment environment. For general cloud users, power is largely abstracted away: consumers simply select the instance type they want and are charged based on time or usage, with power implicitly reflected in the pricing, availability, and performance tiers rather than managed directly. In contrast, for cloud providers and private datacenters, power is a first-class design constraint. Power delivery and cooling capacity often limit how many GPUs can be deployed per rack or per facility, making *performance per watt* a critical metric for maximizing throughput under fixed infrastructure budgets. At the other extreme, in edge and on-device systems, power is the defining

constraint of the entire system. Strict limits imposed by batteries, thermals, and form factors mean that model architectures, precision choices, and execution strategies are fundamentally shaped by power efficiency rather than peak performance. In these scenarios, power determines not only what performance is achievable, but where and how that performance can be deployed in practice.

Comparing the Specs of Popular GPUs

Now that you understand the key GPU specifications, let's explore how to apply this knowledge to determine the best use cases for a given GPU.

Table 5-4 compares the specs and cost of some popular NVIDIA chips for inference. In general, more powerful GPUs will always be more expensive, so your selection really depends on whether the model to be served can benefit from the GPU's increased performance or the specific features it provides.

Table 5-4. Comparing the specs and costs of popular GPUs for LLM inference (these costs were current as of Spring 2026 and may change) (source)

	H200 SXM	H100 SXM	A100 SXM	L40S	A10
GPU memory size (GB)	141	80	80	48	24
FP16 and BF16 TeraFLOPS (Tensor Core)	1979	1979	312	362	125
GPU memory bandwidth (TB/s)	4.8	3.35	1.935	0.864	0.6
Support for FP8	Yes	Yes	No	Yes	No
NVLink/NVSwitch	Yes	Yes	Yes	No	No
Cost on-demand hourly per GPU	~$6.3	~$6.2	~$2.7	~$2.25	<$1.25 or no longer available

Let's look at some examples:

Small model
: If you are serving a fine-tuned model such as Llama-3-8B for a specific use case and latency is not very critical, then A10 might be a decent choice. It already has enough GPU memory to load and execute the model, it's the cheapest option here, and it's highly available, being an older-generation chip.

Mid-sized model
: The deepseek-ai/DeepSeek-R1-Distill-Qwen-14B model has almost twice as many parameters as Llama-3-8B. If you're serving this model, stepping up to the L40S might be a good idea. Especially considering its support of FP8 precision, the L40S could be highly capable for this task. This setup avoids all the overhead of serving models across multiple GPUs; at the same time, it has a good amount of memory left over to achieve long input context, long output generation, and high batch size.

Large model

If you need to serve deepseek-ai/DeepSeek-R1, NVLink Switch becomes a crucial feature to have. This model has 671B parameters, making it too big to fit and too fast for even a single H200 to handle. A likely configuration would be using one machine with eight H200 GPUs connected together in FP8 precision. (The DeepSeek model uses an MoE architecture. This enables further optimizations, such as having experts spread across multiple GPUs or even different nodes. We'll discuss this in the final chapters of the book.)

These are just some simple examples of how different models run on different GPUs. We've introduced quite a lot of new concepts and calculations here, without expanding on them, but don't worry. We will dissect them all in this book and teach you techniques for finding the best solution that meets your serving requirements and reduces the cost-to-serve.

Please note that, in this field, technology changes fast. The latest, hottest chip that everyone wants today can be replaced by a newer one tomorrow, but the foundational concepts you're learning and the intuition you're building for weighing your options do not change.

Now that you have a solid understanding of GPU specifications, it's time to explore the limitations and bottlenecks you may encounter when actually serving LLMs on GPUs. Identifying these bottlenecks is critical, since the optimization techniques introduced throughout the rest of the book are specifically designed to overcome these challenges and improve efficiency.

Bottlenecks in LLM Model Loading

In the following two sections, we will put LLMs into the mix and see how they interact with the GPU specs you learned about in the two previous sections. We will start with model loading, then discuss model execution phases.

We'll look at why models need to be hosted in GPU memory, identify the major consumers of GPU memory, and walk through how to estimate GPU memory requirements for a given model with a given workload.

The Model Loading Process

The first step in efficient LLM serving is to load the model weights and cache them in GPU memory. During the loading process, the model weights are first moved into CPU memory (also called *system memory*) and then transferred to GPU memory, as shown in Figure 5-8. Once loaded, the weights remain cached in GPU memory, ready to serve incoming requests. Thus, it's essential to have enough GPU memory to hold the model weights.

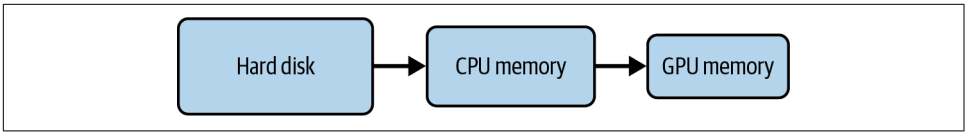

Figure 5-8. Data flow when loading model weights

Some of you may wonder: what if we cached the model in a larger amount of CPU memory, or just load it on demand when serving requests come in? The answer lies in data transfer speeds. Both hard disks and CPU memory are significantly slower at moving model weights, making them impractical for real-time inference. Table 5-5 compares the data-transfer bandwidths of SSD, CPU memory, and GPU memory. Loading from the hard disk would be even slower.

Table 5-5. Comparing data-transfer bandwidth between SSD, CPU memory, and GPU memory

Hard disk (SSD) bandwidth	CPU memory bandwidth	GPU memory bandwidth
0.5 to 14 GB/s	50 to 200 GB/s	300 GB/s to 3 TB/s

It's crucial to distinguish between CPU (system) memory and GPU memory. At a high level, CPU memory and GPU memory are built using different technologies. CPU memory is typically *dynamic random-access memory* (DRAM), optimized for capacity and general-purpose access, while GPU memory is *High-Bandwidth Memory* (HBM), a specialized form of DRAM optimized for massive parallel data movement. The GPU memory is where the model weights should be cached. If model weights are stored inside CPU memory, they need to be transferred to GPU memory to utilize the GPU compute. This extra step introduces an unacceptable delay in serving responses.

Estimating Model Size

How much GPU memory will a large AI model require? When calculating the memory footprint of a model, there are two key variables: the model's parameter count and the data type (precision) of its parameters.

Let's take a model from Hugging Face as an example. A model's number of parameters can often be found directly in its name. For instance, the name Llama-2-7b indicates that this model contains approximately 7 billion parameters in its model weight.

The data type can be found in the *config.json* file in the model's Hugging Face repository (shown in Figure 5-9). Specifically, the `torch_dtype` attribute reveals the precision used for model weights. For example, if the value is `float16` (FP16), this means the model uses 16-bit floating-point precision for storage and computation.

Here is an example of a *config.json* file in Hugging Face, which contains the model configuration and information to correctly instantiate and use the model:

```
{
  "_name_or_path": "meta-llama/Llama-2-7b-chat-hf",
  "architectures": [
    "LlamaForCausalLM"
  ],
  "bos_token_id": 1,
  "eos_token_id": 2,
  "hidden_act": "silu",
  "hidden_size": 4096,
  "initializer_range": 0.02,
  "intermediate_size": 11008,
  "max_position_embeddings": 4096,
  "model_type": "llama",
  "num_attention_heads": 32,
  "num_hidden_layers": 32,
  "num_key_value_heads": 32,
  "pretraining_tp": 1,
  "rms_norm_eps": 1e-05,
  "rope_scaling": null,
  "tie_word_embeddings": false,
  "torch_dtype": "float16",
  "transformers_version": "4.32.0.dev0",
  "use_cache": true,
  "vocab_size": 32000
}
```

The *data type*, or *precision*, of a model refers to the number of bits required to store a single parameter and their corresponding memory sizes. Data type significantly impacts both model size and model performance, because it determines how the model weights are stored. In general, lowering precision—for example, from FP32 to FP16, and then to INT8 or FP8—comes at the cost of some accuracy, but reduces model size and improves serving performance. We will dive deeper into this in the next chapter; for now, you just need to understand the relationship between precision levels, as shown in Table 5-6.

Table 5-6. Parameter precision and memory size

	Bits	Size in bytes
FP32 (single precision)	32	4
FP16 (half precision) / BF16	16	2
INT8 (quarter precision) / FP8	8	1

Now we know that the Llama-2-7b model has around 7 billion parameters that are stored in BF16 format, which means each parameter occupies 2 bytes:

~7 billion parameters × 2 bytes/parameter = 14 billion bytes = 14 GB

Looking at the files for the model confirms this estimate. The screenshot in Figure 5-9 shows the actual sizes of *pytorch_model*.bin* files, which sum up to around 13 GB.

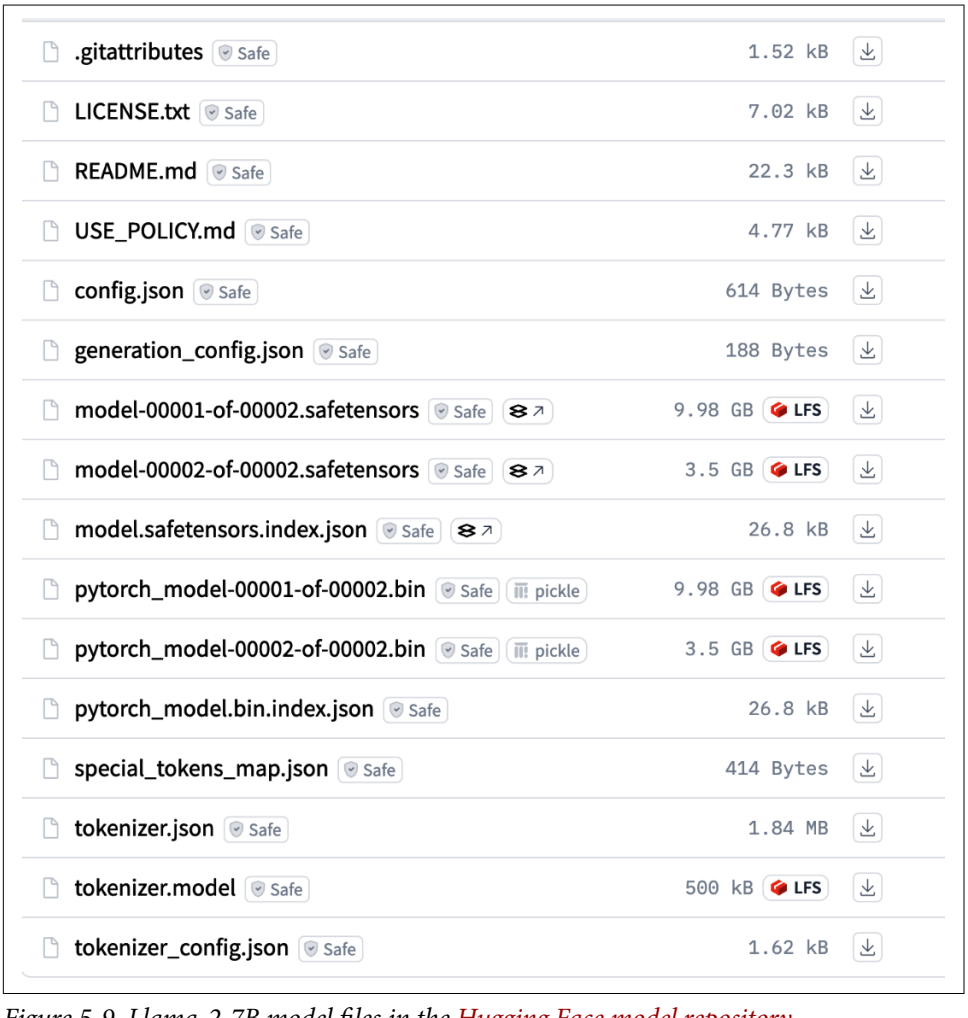

Figure 5-9. Llama-2-7B model files in the Hugging Face model repository

Estimating KV Cache Size

Continuing the example in the last section, we now know the model size, which is around 14 GB. Suppose you have a GPU with 16 GB of memory. You should be able to load it fine, since 16 is more than 14. It will very likely also run fine when you send in one short single request. Yay! But hang on. A GPU with *just enough* memory to hold the model is not ideal.

Recall that in Chapter 2, we dived deep into KV caching, which is an optimization technique that trades some amount of GPU memory space for much faster serving performance by caching the intermediate results. As it decodes the next token, the model can reuse those results without redoing the computation. Figure 5-10 shows an example of how model weight and KV cache co-locate inside GPU memory. A small amount of extra memory should be set aside for intermediate computations. If you only leave a tiny amount of GPU memory for the KV cache, it constrains you to run at very limited batch sizes and with short context. Now let's look at why and see how to estimate how much KV cache space to reserve.

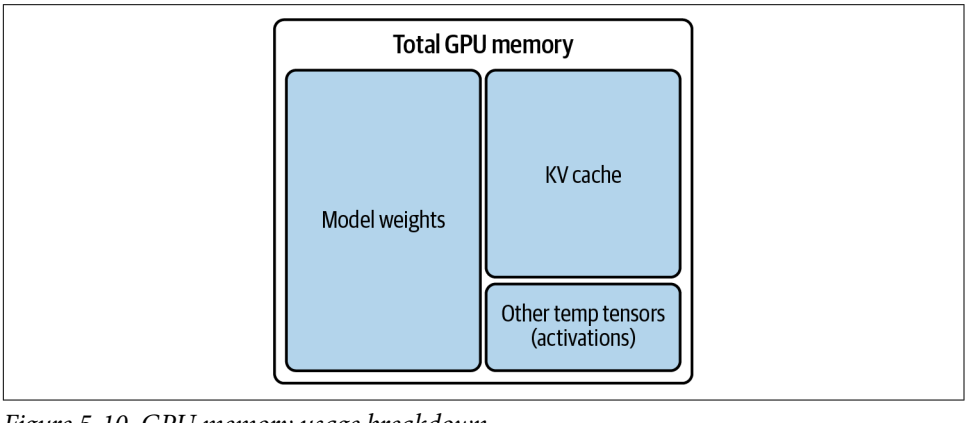

Figure 5-10. GPU memory usage breakdown

The formula for estimating KV cache size for each token is:

> KV cache size per token = 2 × number of layers × number of attention heads × head dimension × data type size

Next, we need to account for the effects of the model's architecture and size. In this simple example, we will use Llama-2-7b, which is still using the basic form of *multi-head attention* (MHA). In later chapters, we will introduce other novel ideas to reduce and compress KV caches such as multi-query attention (MQA), grouped-query attention (GQA), and multi-head latent attention (MLA), which was introduced in DeepSeek V2 and is used in DeepSeek V3 and R1.

Llama-2-7b has 32 attention-head layers and 32 attention heads, with a head dimension of 128 (4096/32). Suppose we still use half precision. Referring to Table 5-6, you can see that the precision comes to 2 bytes per token. Using the formula above, we get:

$$\text{KV cache size per token} = 2 \times 32 \times 32 \times 128 \times 2 = 524{,}288 \text{ bytes} = 0.5\text{MB}$$

We now know that each token requires 0.5MB. The next question is: how many tokens do we need to cache? The maximum number of tokens can be calculated as the maximum batch size times the maximum sequence length. *Max batch size* determines how many parallel users or requests one model instance can process at a time. *Max sequence length* limits the total input length and output length. Thus, if we are processing more requests together and dealing with longer prompts and/or longer generation, the KV cache size will add up:

$$\text{Total KV cache} = \text{KV cache size per token} \times \text{max number of tokens in cache} = \text{KV cache size per token} \times (\text{max batch size} \times \text{max sequence length})$$

Suppose we use the model to summarize long paragraphs, using a max sequence length of 4,096. With a batch size of 16, the calculation would be:

$$\text{Total KV cache} = 0.5\text{MB/token} \times 4{,}096 \text{ tokens per request} \times 16 \text{ requests} = 32 \text{ GB}$$

That's even bigger than the model itself, which is now 14 GB!

Now let's consider two chips, with 24 GB and 48 GB GPU memory, respectively, still with the max sequence length of 4,096. As shown in Table 5-7, using total GPU memory size and subtracting the model size, which is around 14 GB, we can calculate the estimated memory left. We can then derive the batch size by dividing the product of the sequence length (which is 4,096) and 0.5MB per token. In this case, the A10 GPU can only serve 4 parallel requests, while the L40S can serve 16. Even though the L40S is more expensive, it still comes out to be more cost-efficient. (Note, however, that we usually cannot achieve the maximum theoretical batch size from the batch size calculation, because we also need to reserve space inside the GPU for *activations,* which are the tensors created in the intermediate steps.)

Table 5-7. Batch size calculation and cost comparison between two different GPUs.

GPU and memory size	Memory left after model weight loaded	Max possible batch size	On-demand hourly cost per chip from AWS
A10 24 GB	10 GB (24 – 14)	4 ($10 \times 1024/(0.5 \times 4096) = 5$)	$2
L40S 48 GB	34 GB (48 – 14)	16 ($34 \times 1024/(0.5 \times 4096) = 17$)	$3.75

So far, you've learned that during model loading, we want to reserve enough space for both the model weights and KV cache. Model weights need to be cached inside the GPU memory; KV cache space is reserved for execution. Figure 5-11 shows an example of how GPU memory usage changes from idling to execution. During execution, KV cache size will continue to grow as the sequence length grows and we cache more tokens. We need to make sure we have enough memory beyond peak memory usage at the very end of the generation to avoid out-of-memory (OOM) errors.

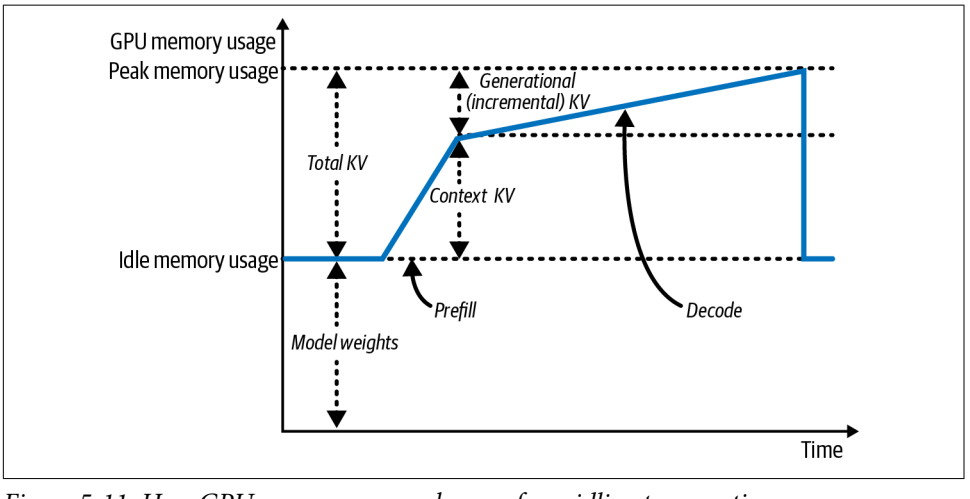

Figure 5-11. How GPU memory usage changes from idling to execution

In later chapters, we will show you techniques (such as prefix caching) that may require more space to achieve faster inference, especially a lightning-fast *time to first token* (TTFT). But as a general rule of thumb, when estimating model GPU memory requirements, we recommend getting started with GPU memory approximately twice the model size to achieve better parallelism and unleash more GPU performance.

Bottlenecks in LLM Model Execution

LLM execution is a complex process. Recall what you learned in Chapter 2: when an LLM processes input and then generates output, it goes through two distinct phases: prefill and decode. These phases exhibit significantly different GPU utilization patterns. To analyze this, we will introduce the concepts of *arithmetic intensity*, which measures computational efficiency, and the *roofline model*, which illustrates hardware limitations and performance constraints.

Boundaries of GPU Compute and Memory Bandwidth

With model loading and its relationship with model size out of the way, now we have the model entirely loaded inside GPU memory and ready for model serving. It is time to discuss the elephant in the room: is the model serving bounded by GPU compute FLOPS or GPU memory bandwidth? In other words, recalling the pizza kitchen analogy, do we need a bigger oven that can bake faster, or do we need to prepare the ingredients and dough faster? Which of the two is the bottleneck in our model serving workload?

To analyze that, we need the concept of *arithmetic intensity*: the ratio of algorithm implementation operations to the number of bytes accessed. This basically calculates the compute (FLOPS) over data movement to the compute unit in a ratio of FLOPS per byte:

arithmetic intensity = number of FLOPS / data movement

If a workload requires very low computation but needs a lot of data reading and writing, then it has a low arithmetic intensity. On the contrary, if a workload does not read and write a lot of data but performs a lot of computations on a small amount of data, it has a high arithmetic intensity.

Now, let's look at data movement, which is often confused with model loading. *Data movement* happens at model execution time, when the model weights are already in GPU memory: the off-chip HBM (recall that HBM is a special type of DRAM). On-chip memory such as L2 cache, L1 cache, and shared memory are built from *static random-access memory* (SRAM) which is smaller and more expensive but a lot faster than HBM, and sits right next to the compute. In other words, SRAM trades capacity and cost for speed, while HBM trades speed for capacity. GPUs use SRAM to support low-latency computation and HBM to store large data such as model weights. During serving, the model weights are moved from GPU memory in HBM, through SRAM, all the way to the register, so the actual compute can execute the operation (Figure 5-12).

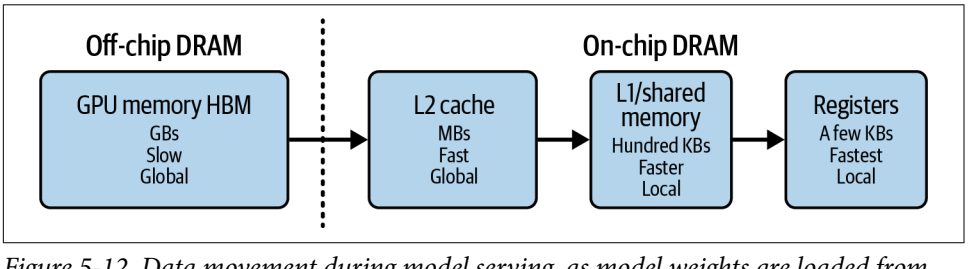

Figure 5-12. Data movement during model serving, as model weights are loaded from off-chip GPU memory all the way to the register for computation

During model serving, the model weights stored inside GPU memory are constantly read, along with all intermediate step outputs, to registers for computation. Since the GPU memory is the slowest of the bunch, GPU memory bandwidth is the number we should use for data movement to the compute unit. (It's a little different if the weights or outputs are small enough to fit into the on-chip caches in a smarter way, such as flash attention, but we'll talk about that in Chapter 6.)

Just by looking at the GPU specs, such as the L40S GPU specs in Table 5-8, we can calculate the chip's theoretical arithmetic intensity.

Table 5-8. Specifications for the NVIDIA L40S GPU

GPU Memory	48 GB GDDR6 with ECC
Memory Bandwidth	864 GB/s
Interconnect Interface	PCIe Gen4 x16: 64 GB/s bidirectional
NVIDIA Ada Lovelace Architecture CUDA Cores	18,176
NVIDIA Third-Generation RT Cores	142
NVIDIA Fourth-Generation Tensor Cores	568
RT Core Performance TFLOPS	212
FP32 TFLOPS	91.6
TF32 Tensor Core TFLOPS	183
BFLOAT16 Tensor Core TFLOPS	362.05
FP16 Tensor Core	362.05
FP8 Tensor Core	733
Peak INT8 Tensor TOPS	733

For half precision (FP16 Tensor Core), we can calculate the theoretical arithmetic intensity for the chip:

$$362 \text{ TeraFLOPS} / 864 \text{ GB/s} = (362 \times 10^{12} \text{ FLOPS}) / (864 \times 10^{9} \text{ B})$$

$$\sim= 419 \text{ FLOPS/B}$$

Now that we have the arithmetic intensity, we can draw a *roofline model*, a visual performance model that relates a system's compute capability and memory bandwidth to help identify whether an application is compute-bound or memory-bound. A *naive roofline model* for the L40S chip, a simplified version that estimates a system's achievable performance based only on its peak compute throughput and memory bandwidth, assuming ideal conditions without accounting for hardware complexities, is shown in Figure 5-13.

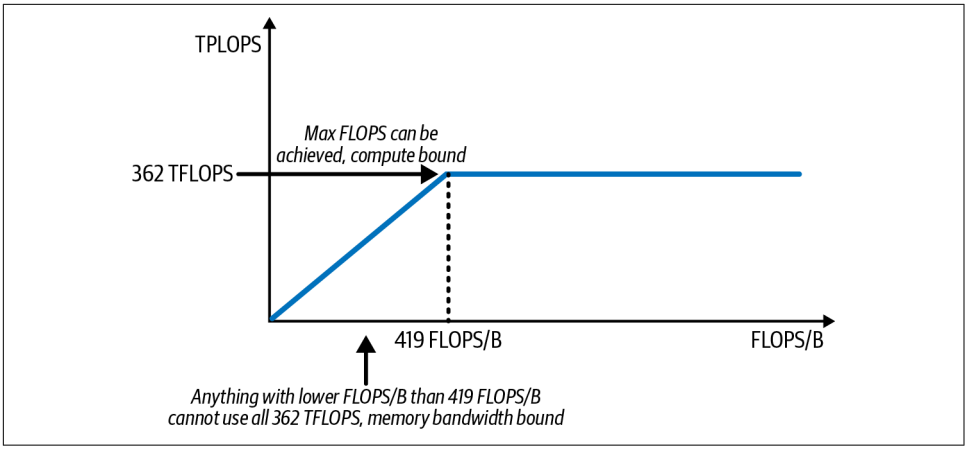

Figure 5-13. Roofline model for the L40s chip

The x-axis shows the arithmetic intensity in FLOPS/B while the y-axis shows tera-FLOPS (TFLOPS). Figure 5-13 shows how much compute the GPU can theoretically do given a certain arithmetic intensity in FLOPS/B. If the arithmetic intensity is lower than the 419 FLOPS/B we calculated, then we cannot use all the 362 TFLOPS the GPU is capable of.

It's time to try using arithmetic intensity and a roofline model to analyze whether a specific workload on a given GPU is bounded by compute or memory bandwidth. Let's say we have a workload to run with ~210 FLOPS/B, basically half of the cross-over point of 419 FLOPS/B. This is depicted in Figure 5-14, data point 1.

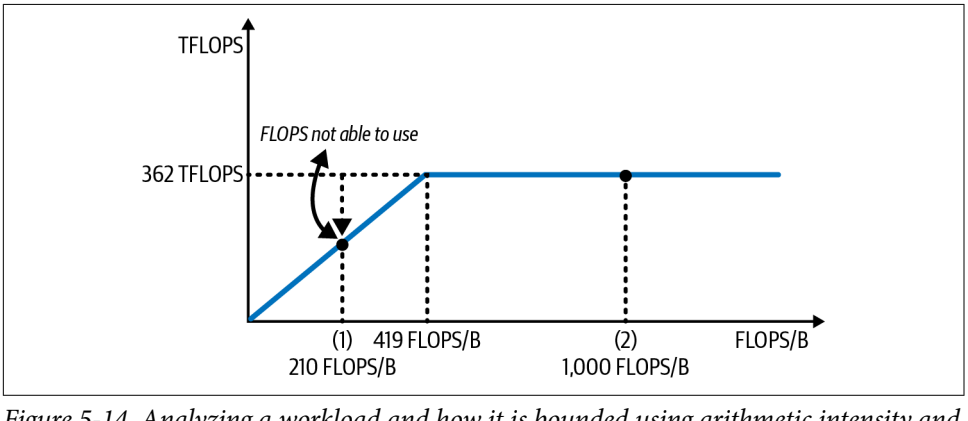

Figure 5-14. Analyzing a workload and how it is bounded using arithmetic intensity and a roofline model

In data point 2, the workload has 1,000 FLOPS/B, which easily hits the maximum TFLOPS the chip can do. In this case, data point 1 is bound by memory bandwidth,

because it cannot utilize the extra compute it has available. (We still have a lot of space in the pizza oven, but we can't supply the dough fast enough.) Data point 2 is considered to be compute-bound because we've already reached the maximum compute available. No matter how much more data we can read, this is the fastest we can calculate using this chip. (We're supplying dough very fast, but the oven has limited capacity and we have to wait for its current load to finish baking.)

Now, with a good understanding of the hardware limitations, we need to figure out what an LLM serving workload looks like. Is it memory bandwidth-bound with low arithmetic intensity, or compute-bound with high arithmetic intensity?

Arithmetic Intensity in Matrix Multiplications

To understand whether the LLM serving workload is compute-bound or memory bandwidth -bound, we need to calculate the arithmetic intensity of the various layers inside the LLM's architecture.

In Chapter 2, you learned that an LLM is mostly made up of Transformer blocks. Inside the Transformer blocks, there are two main components: self-attention layers and feedforward layers. The vast majority of computations for both are matrix multiplications. In LLM architectures, there are many other layers, such as element-wise operations and reduction operations, all of which typically have low arithmetic intensity because the computation on each data point loaded is not as high as in matrix multiplications. But these layers only represent a small portion of the overall computation. So, in this section, we'll first show you how to estimate the arithmetic intensity of a matrix multiplication operation (often shortened to *matmul*).

Matrix multiplication means calculating dot products between rows of the first matrix and columns of the second matrix to produce a new matrix. Figure 5-15 shows an input matrix with shape [M, K] doing a matmul with a weight matrix [K, N], yielding the output matrix [M, N] shown on the right.

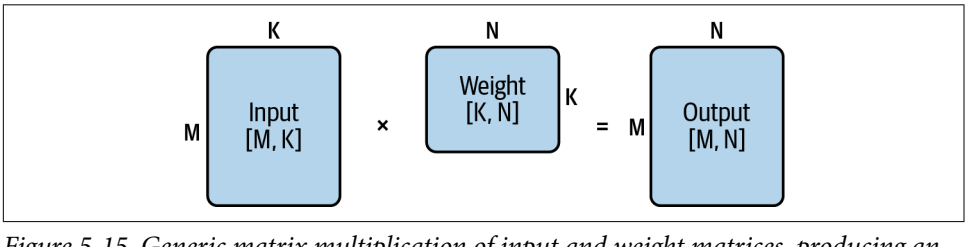

Figure 5-15. Generic matrix multiplication of input and weight matrices, producing an output matrix

The naive code to calculate the matmul is shown in the following in nested triple `for` loops:

```
for m in [0, M):
        for n in [0, N):
                for k in [0, K):
                        Outputs[m][n]+=Inputs[m][k]*Weights[k][n]
```

To calculate the arithmetic intensity, we need to calculate two things: the *numerator*, which is the number of operations performed during the matmul, and the *denominator*, which is how much data is transferred:

Numerator
: For number of operations, based on the preceding code snippet, there are M × N × K multiplications and M × N × (K – 1) additions. We can simplify it to:

$$\text{\# of operations} = 2 \times M \times N \times K$$

Denominator
: For data movement, both input matrices need to be read and the output matrix needs to be written. Assuming the precision of the values inside the matrices are all half precision, with a size of 2 bytes each, we get

$$\text{data movement} = 2 \times (\text{inputs size} + \text{weight size} + \text{outputs size})$$
$$= 2 \times (M \times K + K \times N + M \times N)$$

Thus:

$$\text{arithmetic intensity} = \text{\# of operations} / \text{data movement}$$
$$= 2 \times M \times N \times K / (2 \times (M \times K + K \times N + M \times N))$$
$$= M \times N \times K / M \times K + K \times N + M \times N$$

Using our formula for calculating the arithmetic intensity of a matmul, let's see how the arithmetic intensity changes given different sizes of matrices. Supposing for simplicity that M = N = K, let's calculate a few values for arithmetic intensity, as shown in Table 5-9.

Table 5-9. Calculating arithmetic intensity values for matrices of different sizes

Matrix size (M=N=K)	Arithmetic intensity (FLOPS/Byte)	Verdict based on L40S
64	21	Memory bandwidth-bound
512	170	Memory bandwidth-bound
4096	1365	Compute-bound

In the first two rows, when the matrix size is small, from 64 to 512, the arithmetic intensity is 21 and 170, respectively. Recall the calculation of the roofline model in Figure 5-14: the crossover point for L40S is 419 FLOPS/B. Thus, they are both memory bandwidth-bound. On the contrary, as matrix size grows to 4,096 in the last row, the arithmetic intensity becomes 1365 FLOPS/B—well over the 419 FLOPS/B crossover point. It is now compute-bound.

As you can now see, the operation can be pushed to be compute-bound if the matrices in the matrix multiplication operation are large enough. Then the question in LLM serving is: are the matrices large enough? You might think, given the size of the model, that they should be, right? Not always. Let's break that down in the next section.

Applying Arithmetic Intensity Analysis to the LLM Prefill and Decode Phases

You now know that the bigger the compute matrix, the higher the computational intensity during matrix multiplication. However, it's not always possible to achieve big matrix multiplications in LLM serving. This is especially true when the request batch size is 1, which means the model is only processing a single request at the time. (We will discuss batching in Chapter 6.)

Let's dive deeper into the shape of the input data passed to the model. The "shape" here describes the dimensions of the data, commonly referred to as its *tensor*. In this case, our input tensor is three-dimensional and has a shape of [batch size, sequence length, model hidden dimension]. *Batch size* is the number of requests we are processing at once. *Sequence length* is the number of tokens in the input prompt. From simple phrases such as "Hello" to big paragraphs a million tokens long, the sequence length is highly variable and mostly up to the user. *Model hidden dimension* is the number of values each token's vector has in the model, impacting its capacity and performance. Since we are now only considering one request, batch size is set to 1. Consequently, the three-dimensional input tensor can be reduced to a matrix that has only two dimensions [sequence length, model hidden dimension].

Before we start the arithmetic intensity analysis, let's revisit the two main phases of model serving from Chapter 2: prefill and decode. The *prefill* phase is the initial processing stage, where the model encodes the input prompt before generating any output tokens. During this phase, the model does two things. First, it computes attention over the input tokens using its Transformer layers; second, it generates the first set of KV cache entries and stores them for the subsequent generation phase. In the next stage, the *decode* phase, the model generates tokens one by one using the previously computed KV cache.

Now let's analyze the arithmetic intensity of the prefill and decode phases by looking at their matrix multiplication. You will see the difference in Figure 5-16.

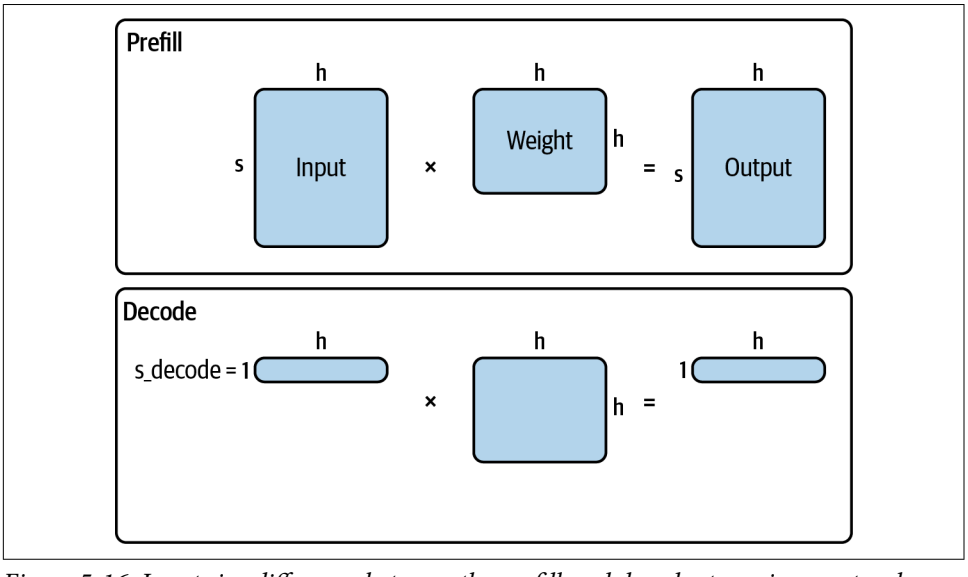

Figure 5-16. Input size difference between the prefill and decode stages in a matmul operation

In Figure 5-16, *M*, *N*, and *K*—which we used in Figure 5-15 for generic matrix multiplication—are replaced with *h* and *s* to demonstrate the actual shape of the matrices in LLM serving. The variable *h* corresponds to the weight (or "hidden") dimension, which does not change between prefill and decode. In prefill, because all tokens are processed together, the matrix has a large dimension for the number of rows in *s,* which is the sequence length of the input. During decode, however, because tokens are generated one token at a time, `s_decode`, which denotes the sequence length during decode, is only 1.

Next, in Table 5-10, let's plug the numbers back into the arithmetic intensity formula, which again is:

$$\text{arithmetic intensity} = M \times N \times K / M \times K + K \times N + M \times N$$

$$= s \times h \times h / s \times h + h \times h + h \times h$$

Table 5-10. Arithmetic intensity comparison between prefill and decode at different sequence lengths

Sequence length M = s for prefill M = 1 for decode	Hidden dim K = N = h	Prefill arithmetic intensity (FLOPS/B)	Decode arithmetic intensity (FLOPS/B)	Verdict based on L40S
64	4096	31	0.5	Memory bandwidth-bound for both prefill and decode
512	4096	240	0.5	Memory bandwidth-bound for both prefill and decode
4096	4096	1365	0.5	Compute-bound on prefill but memory bandwidth-bound on decode

From the numbers we get for prefill and decode calculations, you can see that prefill can have a high enough arithmetic intensity to saturate GPU compute, making prefill compute-bound when the sequence length is high. However, no matter how long the sequence is, decode will always be very low, making it memory bandwidth-bound.

In summary, different components such as prefill versus decode, and different layers of the model, can all have very different arithmetic intensity. Arithmetic intensity is also affected by sequence length (as well as batch size, which we've intentionally skipped for now). All these analyses and calculations are designed to help you develop your intuition about bottlenecks in the different phases of LLM serving, so you'll later be able to understand why various optimization techniques work and how to select them. For example, to improve performance, if a workload is compute-bound, you should explore ways to optimize mathematical computations and reduce FLOPS. If it is memory bandwidth-bound, you need to minimize unnecessary data movement.

Other AI Accelerators and Trends

So far, we have been solely investigating NVIDIA GPUs, but the core concepts for how the model loads and executes are the same on other chips.

In the LLM inferencing space, a lot of other chips are competing with NVIDIA, some of which are designed specifically for model inferencing. GPUs such as AMD's MI300X and Intel's Gaudi2 are both alternatives to NVIDIA's H100 and A100. Google's TPU and Amazon's Inferentia are mostly only available on their respective clouds. Huawei's Ascend NPU is another notable competitor, particularly in markets where NVIDIA faces restrictions. And there are many startups in this space, such as Groq, Cerabras, Untether AI, SambaNova, and d-Matrix.

As we write this in early 2026, NVIDIA's GPGPU solutions still dominate this market. There are several reasons for its dominance:

- Some of these offerings are specifically tailored for deep-learning-based model inference, matrix multiplication workloads, or Transformers. Their architectures are simpler, more cost-efficient, and more energy-efficient, but compared to GPGPU, they often struggle with rapidly evolving demand from model architectures.
- At the software level, NVIDIA GPUs have much more mature support with the CUDA ecosystem, which is widely used in both training and inference. Custom chips usually require their own proprietary software stacks, such as ROCm for AMD GPUs, JAX for TPU, and Neuron for Inferentia. Switching over can be costly, especially given the lower levels of community support.
- Some chips leverage on-chip SRAM to achieve impressive performance with superb latency. However, so far, this solution is rarely very cost-effective, given the high price of SRAM and the number of chips needed to serve one model instance.
- NVIDIA GPUs still have the edge in flexibility to meet different inference setups. They support different levels of precision, such as FP8, FP4, high memory bandwidth, and advanced GPU interconnect capability, which are all contributing factors in this area.

While all these factors contribute to NVIDIA's continued dominance in the market, the broader landscape of hardware acceleration is evolving rapidly. As new architectures emerge, one of the biggest challenges is overcoming bottlenecks, since data movement has not kept pace with improvements in raw compute power.

Figure 5-17 compares the increases in hardware compute FLOPS, memory bandwidth, and inter-GPU bandwidth over the last 20 years. You can see that compute capability has improved a lot over time, and at a much faster pace than data movement speed, in both memory bandwidth and inter-GPU bandwidth. Nowadays, GPU data-movement limitations have become an obstacle to utilizing the fast advancements in compute. This problem is usually referred to as the "memory wall." Quite a lot of accelerator-chip manufacturers are trying to tackle this issue.

As the gap between compute and data movement continues to widen, accelerator designers are increasingly focusing on architectural approaches that mitigate the memory wall. Two notable trends have emerged in recent AI hardware designs.

First is pushing more computation closer to on-chip SRAM. With large amounts of on-chip SRAM, chips following this design philosophy aim to keep model parameters and intermediate data as close to the compute units as possible, significantly reducing memory-access latency.

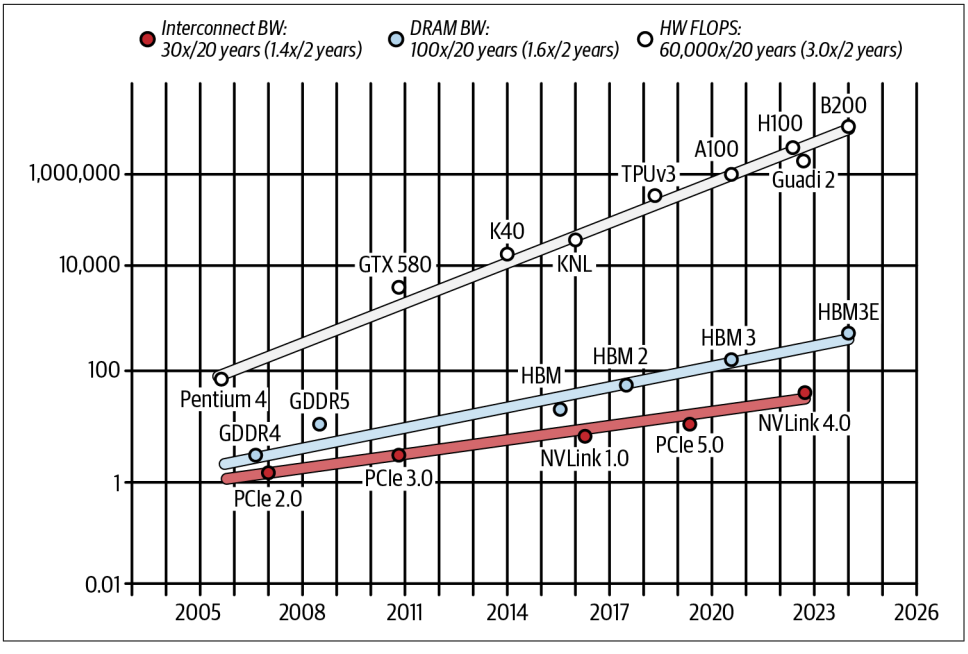

Figure 5-17. Scaling in peak hardware FLOPS, and memory and interconnect bandwidth (source: An et al., 2024)

This expensive design can achieve extremely low and predictable latency, making them very attractive for latency-sensitive inference workloads. A concrete example can be seen in the recent collaboration between Groq and NVIDIA. NVIDIA's interest in this approach highlights a broader industry recognition that, for certain inference workloads, extreme low latency can be more valuable than peak throughput, even if achieving it requires higher silicon cost and more careful model partitioning across chips.

The other important trend is scaling performance through tightly coupled multi-GPU systems as a whole. This trend also extends to the CPU layer, where designs such as NVIDIA's Grace CPU are tightly integrated with GPUs via high-bandwidth interconnects, reducing CPU–GPU data-transfer overheads in large-scale multi-GPU systems.

A representative example of this system-level approach is the GB200 NVL72, NVIDIA's rack-scale architecture built around tightly coupled CPUs, GPUs, and interconnects. Let's dissect it step by step. Starting from a single GPU, the system relies on the B200 GPU where "B" stands for Blackwell generation. The "G" in front of B200 yielding GB200 means it is built with NVIDIA's Grace-Blackwell building block where Grace CPUs are connected with Blackwell GPUs via NVLink-C2C to achieve high-bandwidth, low-latency CPU–GPU coupling. At the end, NVL72 is the rack-

scale system that connects 72 Blackwell GPUs into one large NVLink domain using the NVLink Switch System. In total, GB200 NVL72 connects 36 Grace CPUs with 72 Blackwell GPUs in the rack-scale design. This rack-scale architecture is particularly powerful when combined with disaggregated serving for large MoE models, which is an advanced technique we will be covering in Chapter 7. So far, for these very latest setups requiring a strong software–hardware codesign, the majority of the industry is still in the progress of picking them up and only a few hyperscalers such as Frontier Labs and big tech are at the forefront of this adoption.

Together, these approaches address the memory wall both within the chip, using fast on-chip SRAM, and across chips, using tighter, higher-bandwidth GPU–GPU and CPU–GPU interconnects.

Summary

This chapter began by showing you why LLM serving is crucial for business applications. Efficient LLM serving can improve customer experience, save costs, and even determine whether the business can be sustained and expanded.

Next, we turned to understanding AI accelerators, focusing on the key GPU specs of memory size, compute FLOPS, and memory bandwidth. You also learned why intra-node and inter-node GPU communication technology can be important for large models when a single GPU is not enough. Additionally, we touched on the current state of other AI accelerators and how they are trying to overcome current limitations on GPU advancements, known as the "memory wall."

With that understanding of the hardware side, we started to show you how an LLM runs on the hardware in both the model loading and model serving phases. We introduced arithmetic intensity as a key approximation technique in estimating model performance and identifying if it is bottlenecked by compute or memory bandwidth.

After some concrete examples with simplified calculations to illustrate the change in arithmetic intensity during prefill and decode, we concluded that LLM serving has both compute-bound phases and memory bandwidth-bound phases. When batch size is fixed at 1, LLM can be very memory bandwidth-bound, especially when decode or the input sequence is not long enough. In this case, the GPU compute is not saturated enough to reach its full potential.

In this fast-evolving field, new techniques come out at an unprecedented pace. But the intuition you built from analysis will equip you to better understand them and adopt them if they fit your situation. Even if language models switch to a new architecture or you start working on vision models, your knowledge about hardware specs, how models run on hardware, and the general intuition will remain valuable.

CHAPTER 6

Essential LLM Optimization Techniques

In prior chapters, we demonstrated the importance and the challenges of optimizing LLMs for serving. In the next two chapters, we will dive deep into each of the critical LLM optimization techniques one by one so that you are equipped with the knowledge to decide when, how, and why to use them for your serving needs.

In this chapter specifically, we will focus on essential techniques that will help you understand most optimization concepts and achieve a lot of your optimization goals. We'll leave the more advanced techniques and industry trends for Chapter 7.

In this chapter, we will discuss how to use:

- Request batching and scheduling to achieve better parallelism and GPU utilization
- Attention optimization to achieve better compute efficiency, less required compute, and better memory management
- Model compression to achieve smaller models, less memory movement, and/or less compute
- Prefix caching to cache and reuse prior prompts, including how to do it efficiently and obtain a high cache-hit rate

Request Batching and Scheduling-Level Optimizations

In Chapter 2, we divided serving into offline serving and real-time online serving. In real-time online serving, requests are received as the user sends them, while for offline serving, we have the requests already and can batch them all together so that they form a big tensor input that gets fed into the model, instead of sending them one by one.

Grouping requests together during serving can achieve higher throughput, even though it can mean slower responses. Let's dive a bit deeper here to understand why, using a concept you learned in Chapter 5: *arithmetic intensity*.

Why Do We Need Batching in Real-Time Serving?

Recall that LLM serving has two phases, as first explained in Chapter 2: prefill and decode. The prefill phase is when the model understands the input prompts. Tokens in the input prompt can be processed in parallel and achieve high arithmetic intensity, as analyzed in Chapter 5, leading to a compute-bound workload. The decode phase is when the LLM generates one token at a time. Because of this phase's autoregressive nature, Decode is *memory bandwidth-bound*: the model basically needs to go through all of its billion level parameters to generate only one token at a time, which is very inefficient in terms of GPU memory bandwidth (measured in FLOPS) (Figure 6-1).

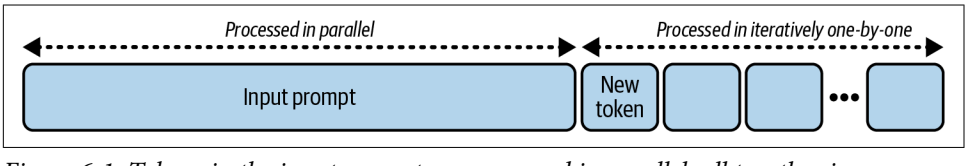

Figure 6-1. Tokens in the input prompt are processed in parallel, all together in one go, during prefill, while new tokens are generated one by one during decode

To fully utilize the GPU compute, you can add more requests with batching and process them together. Let's add that into the setup shown in Figure 6-1. We'll suppose a max batch size of 3 for all the figures in this section.

We'll take three input prompts, `prompt1`, `prompt2`, and `prompt3`, batch them together, and send them to the model. The decode step still only generates one token per iteration, but since we are batching these requests together, we can generate three new tokens in one go: one for each request, as shown in Figure 6-2. This artificially increases the arithmetic intensity: we are still reading the model weights once but doing more calculations and generating more tokens.

In summary, batching is especially effective during the decode phase, where the model generates one token at a time. By grouping multiple requests, batching increases overall throughput and improves GPU FLOPS utilization. In contrast, its benefits are limited during the prefill phase, during which the model already processes all input tokens in parallel. As long as the input prompt is not very small with very limited tokens (say, less than 1,024), prefill easily saturates GPU compute capacity on its own, so additional parallelism from batching has minimal impact.

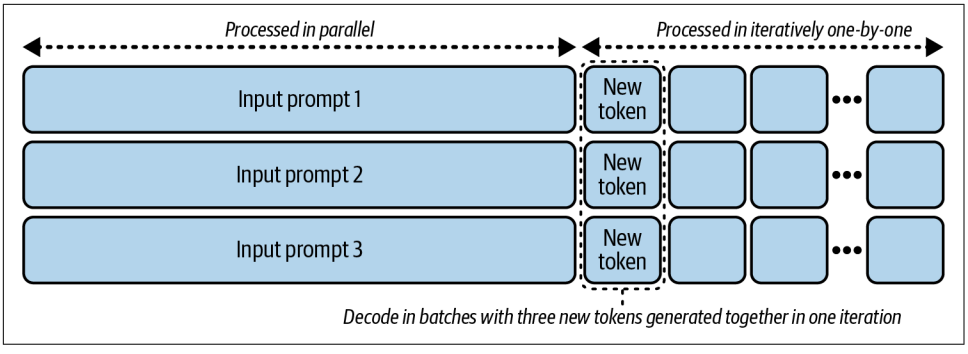

Figure 6-2. With three batched requests during decode, three new tokens can be generated all together

Dynamic Batching in Online Inference

The example we just gave you showed the benefits of batching, but in real life for online inference, you won't have a list of prompts you can process beforehand. So how do you achieve batching? One way is through *client-side batching*, where the client tries to accumulate requests on its end and send them all together once it reaches a preset batch size. We can do something similar on the server side: having the server wait for incoming requests to fill the preset batch size completely before processing them. This is called *static batching*.

Both methods are good for offline use cases, but not ideal in online inference. While offline batching is very common and straightforward, online batching is much more complex. This is because real-world requests are random and can arrive with big time gaps between them, which means filling a batch to maximum can take a long time, resulting in high latency. For example, suppose we have a batch size of 10. The first 9 of those 10 requests arrive within a single second, but the 10th request arrives 5 minutes later. The first 9 requests would not be processed until the 10th arrives; they would just wait there for 5 minutes.

The solution is *dynamic batching*, which groups incoming requests together at inference time based on defined logic, to balance latency and throughput dynamically. This logic usually uses two key parameters: *batch size* (also called *max batch size*, *preferred batch size*, or *max number of sequences*) and *max delay time*. *Batch size*, as you know, determines how many requests can be grouped together before they are sent to the model. *Max delay time* is the maximum time that existing requests can be held while waiting for the other requests to fill the max batch size. If the pending request count reaches the max batch size, the batch is sent immediately, even if the max delay time is not yet met. Similarly, if the max delay time is reached, the whole batch is sent immediately—even if there is only one request in it.

We can make an analogy here. Imagine you're managing a small ferry-boat business. Each boat can carry up to 10 people across the river at a time. People arrive at the dock at different times, and your job is to minimize their waiting time while also trying to avoid sending half-empty boats. Without batching, you'd be sending a boat across every time someone arrives: basically one person, one boat. The passengers would have a great experience, but it would be a horribly inefficient and expensive way to run a business. With static batching, the boat would wait at the dock until 10 people showed up to fill it completely. This would be efficient, with minimal waste, but if people arrived slowly, the very first arrival might have to wait for a long time. That kind of latency would be just as bad for a ferry-boat business as it is for LLM serving. Dynamic batching tries to find a good middle ground that balances latency and throughput: the boat's max batch size would still be 10, but you'd also set a max delay time of 5 minutes. If only 8 people arrive within that 5 minutes, the boat can leave without further waiting. If 10 people arrive before the 5-minute mark, the boat is sent right away.

Now that you know the benefit of dynamic batching, you also need to understand how to tune these two parameters—max batch size and max delay time—based on your latency and throughput requirements. Generally speaking, you want to achieve as high a batch size as possible while still meeting the latency SLA. High batch sizes aren't always possible, because as you increase batch size, the processing latency will go up, and the memory usage of the GPU or CPU will also increase until it runs out of memory. Max delay time is another one to tune. Too long a delay time combined with high batch size would force requests that have already arrived to wait for an extended period of time; too short a delay time would prevent you from filling the batch in time and reduce the actual batch size sent for processing.

Continuous Batching for LLM Online Inference

Dynamic batching works pretty well for most models and has been adopted for many traditional ML serving settings. However, LLMs pose a bigger and unique challenge: LLM serving can have varying input and output lengths, which causes requests in a batch to take vastly different amounts of time to process. In dynamic batching, the total time it takes for the batch to finish is determined by the longest and slowest request, since the result is returned when all requests in the batch complete. Figure 6-3 is a simplified example where we abstract out prefill and decode, just to show you that because the requests have different lengths, one very long request can cause a lot of idle time.

In the boat-business analogy, let's assume your boat needs to take people directly to their homes on the other side of the river. Their trip times could be very different depending on where they live, just like LLM requests having different lengths. The boat has to drop off the person whose home is the furthest away before returning to

the dock for the next batch of people, which means it will run with lower and lower capacity.

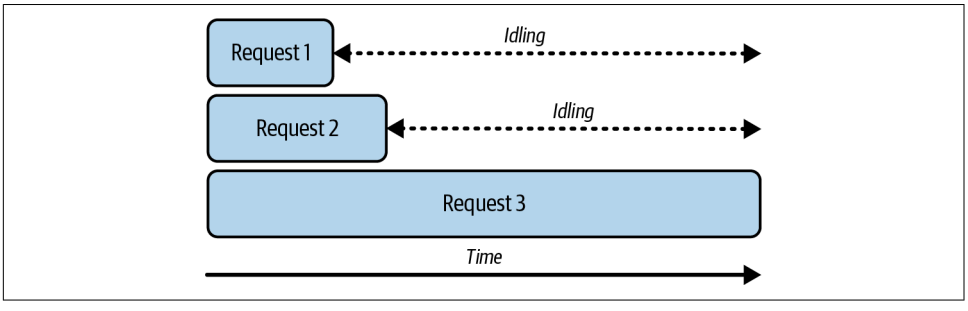

Figure 6-3. The uneven lengths of the three requests can cause significant GPU idle time in LLM serving, as requests 1 and 2 wait for request 3 to finish

In order to mitigate this unique challenge, *continuous batching* (also known as *inflight batching* or *iterative batching*) for LLM serving was introduced. While dynamic batching has a wait time and processes requests batch by batch, continuous batching adds the requests to the backend model and groups them on the fly, without waiting for a fixed batch size or time window. In other words, as soon as one running request in the batch finishes, the queued request gets added to the batch immediately.

In Figure 6-4, you can see that at the very beginning, requests 1, 2, and 3 are sent to the model for processing. Once request 1 finishes, a continuous batching strategy immediately adds the newly arrived request 4 into the processing. The same goes for the newly arrived request 5 when request 2 finishes, as well as request 6 when request 5 finishes. With dynamic batching, newly arrived requests 4, 5, 6 would all be waiting for request 3 to finish before they are dispatched for processing.

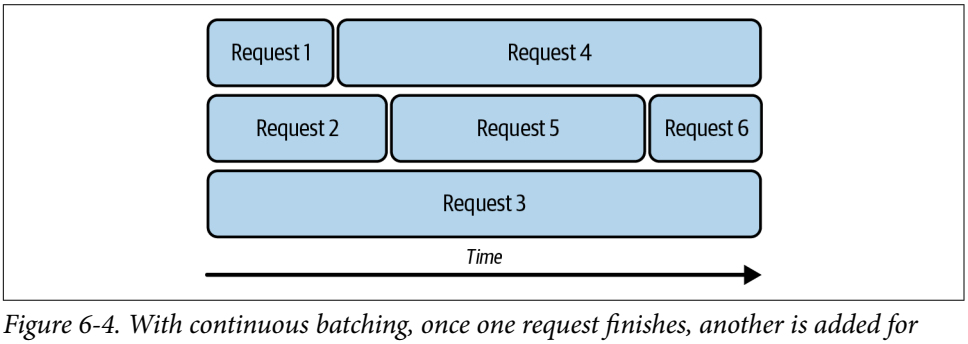

Figure 6-4. With continuous batching, once one request finishes, another is added for processing to minimize GPU idle time.

In the boat analogy, you have one boat that can hold a max of 10 people, and you must carefully manage this one boat to make sure it does not wait too long or run while empty. Now, what if you replaced that single 10-person boat with 10 small boats that can each hold one person? This way, once a person arrives, one boat could leave immediately. Even if the people have different homes at different distances, there is no waste: the boat just comes back once it transports that person and picks up the next person waiting. This sounds like the perfect scenario!

With continuous batching, the next request in the queue is sent for processing once one prior request is complete. This means you don't need to set a max delay time artificially, like in dynamic batching. You'll still need to manage and tune the max batch size, though.

Similar to dynamic batching, higher max batch size means more requests are allowed to be processed in parallel. However, this parameter is only used as an upper bound to avoid going over the absolute max batch size during the continuous batching process. In many modern LLM serving frameworks nowadays, another parameter is added to control whether a request is queued or added to the batch: the *max number of batched tokens*. While max batch size controls things at the request level, basically determining the maximum number of requests the model is processing at a given time, capping the number of batched tokens provides more granular control at the *token* level. That's important because input lengths for LLM serving can vary a lot. For example, batching 10 requests with an input length of just 20 tokens each would be a very different workload than batching 2 requests with 100,000 tokens each. Solely relying on the request level limit is not enough to take different token lengths into account and can lead to batching together too little or too much workload.

To better illustrate the relationship between these parameters, see Figure 6-5.

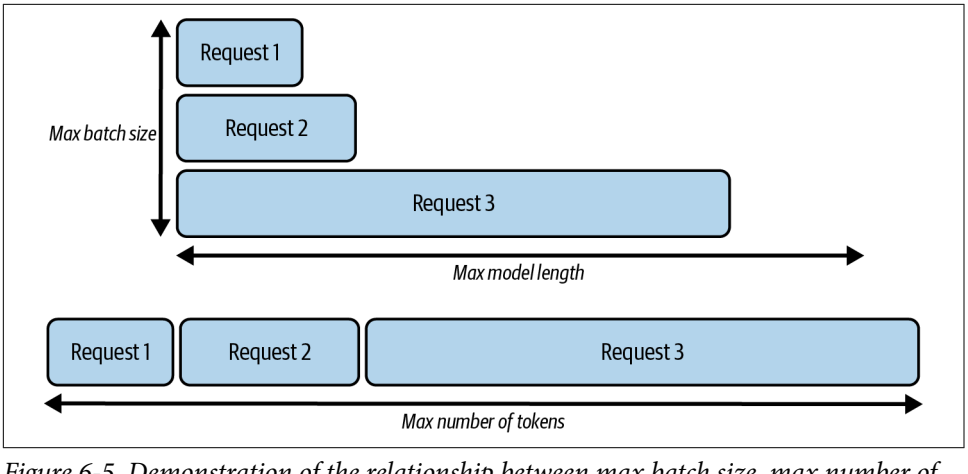

Figure 6-5. Demonstration of the relationship between max batch size, max number of tokens, and max model length

Here, the max batch size is set to 3, so in total, 3 requests can be processed all together in a batch. The length of the bar for each request represents the number of tokens, which should be less than the *maximum model length*, also known as *max context size*, of the model (discussed in Chapter 5). The maximum number of tokens controls how many tokens in total the scheduler allows to be batched. If the scheduler receives a few very long requests that easily hit the max number of tokens limit, it will stop more requests from joining the batch.

With these two parameters, max batch size and max number of tokens, working together and requiring us to meet both sets of constraints, we can fine-tune how many requests can be processed across the prefill and decode phases. In the prefill phase, the key constraint is the maximum token count, since inputs are much longer. In the decode phase, parallelism is typically capped by the maximum batch size. It's important to set the max number of tokens high enough. If it is too low, we might not saturate our GPU compute, because we won't be sending enough tokens for the GPU to process in parallel during prefill. Example 6-1 provides code to configure these parameters in vLLM.

Example 6-1. Code snippet to configure max number of batched tokens (`--max-num-batched-tokens`) and max batch size (`--max-num-seqs`) in vLLM

```
vllm serve \
 Qwen/Qwen2.5-7B-Instruct \
 --max-num-batched-tokens 4096
 --max-num-seqs 128
```

Continuous Batching with Chunked Prefill

Continuous batching is great for solving the problem of requests having different lengths, but we are overlooking another unique aspect of LLM serving: the prefill and decode phases constitute two different workloads. In the last section, when we discuss dynamic batching and continuous batching, we abstracted away the prefill and decode phases in the diagrams and solely used bars to represent request lengths. Now it is time to take a deeper look at how continuous batching works and see if there are opportunities to improve further.

Recall that in LLM serving, prefill is the phase where the model first processes the input prompt. This phase has high arithmetic intensity and does not require much help from batching to be efficient to run on GPUs. Decode is the phase where the model starts generating new tokens in an autoregressive manner; it usually has low arithmetic intensity and can benefit a lot from batching.

When we discussed the benefit of batching, we used a diagram similar to the one in Figure 6-6, which is a cherry-picked "happy path" scenario in which everything lines up perfectly: request input prompts all have the exact same length, generate the

same number of tokens, and start running at the same time. In the first iteration, the prefill steps from all three requests are batched together. Upon finishing that batched iteration, the model starts the second batched iteration, which contains the decode steps from each of the three requests running together.

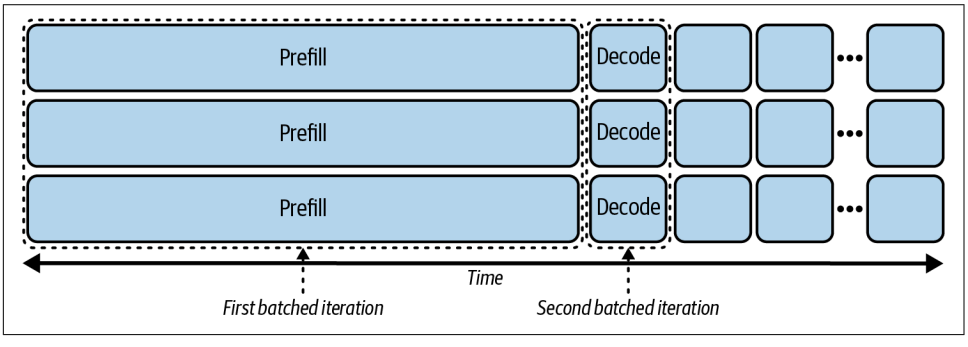

Figure 6-6. A cherry-picked "happy path" scenario where all requests have prefill and decode lined up perfectly

However, in the real world, not only are requests' input and output lengths random, but with continuous batching, the start times are also different. Requests arrive at different times and we process them as soon as we can. So here's the question: if the first request is already running in the decode phase and the second request arrives and wants to start prefill, do we prioritize the decode of one request or the prefill of another request? Can we batch them together in the same iteration?

Let's start with the solution of *not* batching prefill and decode together, since running a hybrid workload often requires more complex GPU kernels. In this way, as Figure 6-7 shows, we only receive one request (request 1). We performed two iterations: iteration 1 to finish the prefill phase and iteration 2 to perform one decode step. Then requests 2 and 3 come in. Now we need to decide whether to prioritize prefill or decode. Usually, we choose prefill, because prefill determines the *time to first token* (TTFT), an important latency metric—especially for interactive usages such as chatbots. However, when we're doing prefill for requests 2 and 3 in iteration 3, request 1 will be waiting there completely idle. If the prompts for requests 2 and 3 are long, like those shown in iteration 3, it really hurts the end-to-end latency and inter-token latency of request 1.

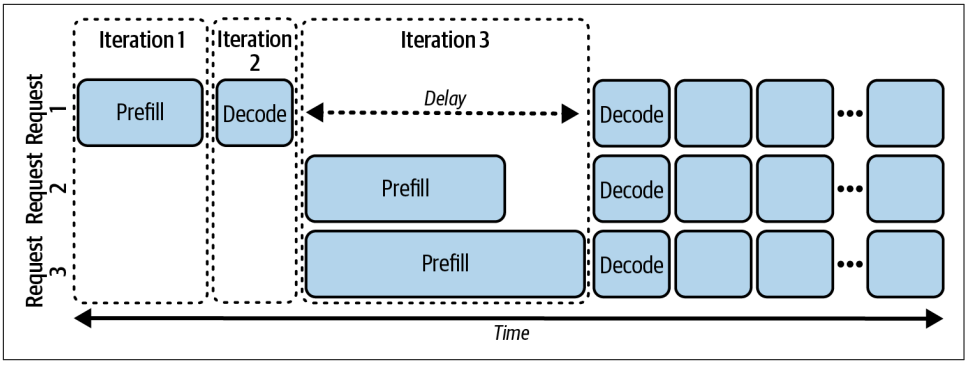

Figure 6-7. Continuous batching with no prefill and decode, batched together and prioritizing prefill

Now, let's consider combining prefill and decode together, as shown in Figure 6-8. Again, the second decode step of request 1 is moved into iteration 3, to run with the prefill steps of requests 2 and 3 in the same batched iteration. However, this still doesn't help much. The delay is still quite prominent, because decoding one token can go much faster than finishing the prefill step, especially when the input prompt is long.

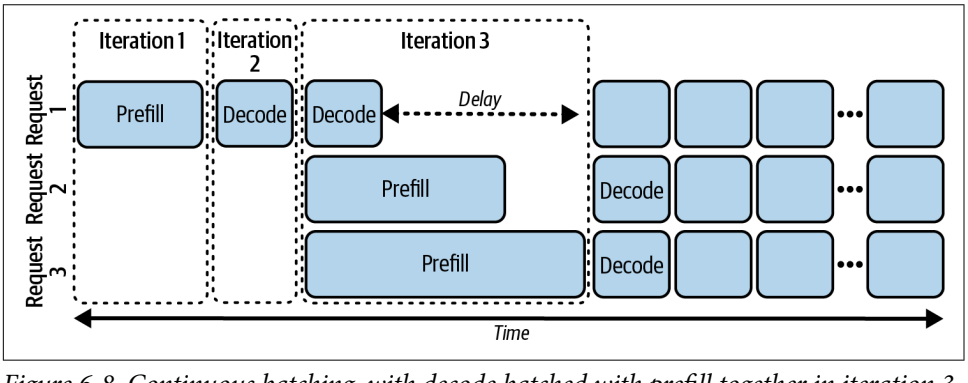

Figure 6-8. Continuous batching, with decode batched with prefill together in iteration 3

One solution to mitigate this is called *chunked prefill*, a technique that splits the long input prompt into smaller chunks. As shown in Figure 6-9, all the prior long prefill bars are now split into several smaller prefill boxes with similar sizes to the decode boxes (ideally), meaning they have similar processing times. As requests 2 and 3 join the batched iteration, request 1 continues to do the decode step, while the other two requests start their own smaller chunked prefill step, as shown in iteration 5. Later, in iteration 10, request 2 seamlessly transitions to decode, since it has a shorter prefill than request 3.

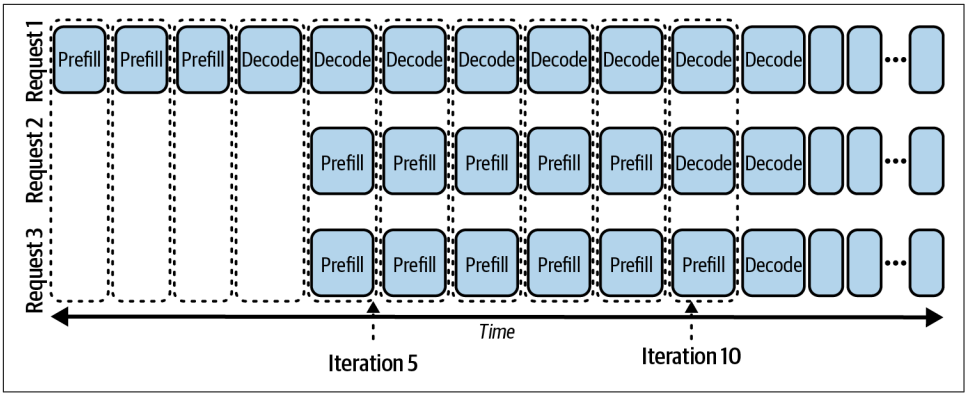

Figure 6-9. Continuous batching with chunked prefill

The benefit of chunked prefill is that even when the input prompt is long, because it is chunked into smaller ones, decode boxes are no longer waiting there, blocked by the long prefill. This improves *inter-token latency* (ITL), which we discussed in Chapter 4. However, it gains ITL at the cost of prefill performance, since it adds more workload during prefill, which makes the TTFT longer. Chunked prefill doesn't really improve end-to-end latency either; in fact, it usually has some negative effects due to the overhead of computing multiple smaller prefill steps. So this is a trade-off you need to make based on your use case and SLA requirements. On the other hand, throughput usually improves with chunked prefill, because it improves batch efficiency by filling gaps of idle time to better utilize the GPU.

Another parameter you need to tune is how many smaller prefills you split the original long prefill into—in other words, how many tokens you want your small prefill jobs to handle. This is also part of the max number of batched tokens. If you choose an extremely large number of tokens, chunking is heavily reduced—for example, taking it all the way to the max model length would mean not doing any chunking of the prefill at all. Too few tokens in each small prefill would mean more overhead, which is not good either: we wouldn't be batching enough tokens in one iteration to saturate GPU compute. Ideally, we would find a value in the middle: not small enough to cause a lot of overhead and low GPU utilization, and not large enough to defeat the purpose of doing chunked prefill.

Continuous batching has been the industry standard in production-level LLM serving for a few years as of this writing. Chunked prefill and its variations are also very popular for many use cases, such as handling long context workloads. One even more advanced technique, called *prefill–decode disaggregation*, completely breaks down the prefill and decode jobs into different GPUs and even different nodes. We will discuss this in the next chapter, after building more basic knowledge in this chapter first.

Scaling Attention and Kernel Optimization

Recall from Chapter 2 that during LLM serving, the two major components in a Transformer block are the attention and feedforward layers (FFN, also called the *multilayer perceptron* or MLP layers). In this section, we focus on how to scale the attention aspect of LLM serving. We will start with reducing KV cache size, then discuss how custom GPU kernels improve attention computation and memory access.

The *attention* mechanism is the key to the breakthrough of LLMs. Yet the original multi-head attention formulation that powers GPT-3 is no longer the only or the most efficient way to leverage queries, keys, and values at scale. Production workloads and the high compute cost of LLMs force us to squeeze more performance out of models, spawning the creation of *multi-query attention* (MQA), *grouped-query attention* (GQA), and the latest, *multi-head latent attention* (MLA), to slash KV cache size while preserving model quality.

After we look at attention optimization, we will discuss the GPU kernels that execute attention. Over time, these kernels have advanced from generic kernel fusion to attention-specialized, hardware-tuned, cache-aware kernels. We will start with understanding kernel fusion and then go into specific kernels, such as FlashAttention.

At the end of the chapter, we'll look at another key innovation, PagedAttention, which introduces a novel mechanism to manage how KV cache is saved in GPU memory to achieve high memory utilization.

Scalable Attention Mechanisms

Reducing the KV cache size is one key way to improve LLM serving performance. During the decode phase of LLM serving, KV cache is constantly transferred per decode iteration from HBM into on-chip registers and shared memory. Smaller KV cache size alleviates pressure on GPU memory bandwidth, as you learned in Chapter 5. Another benefit is that a smaller KV cache also means less space is used in GPU memory, which means you can run more requests in parallel with higher batch sizes for more throughput, and serve longer context than was previously possible with limited GPU memory space.

Starting with the very left of Figure 6-10, *multi-head attention* (MHA) is the original version that underpins many earlier models. But in this original setup, for each query, calculating its attention requires one distinct key and value head. This makes the KV cache for that model architecture the largest and the least efficient of all four of the setups we will discuss here.

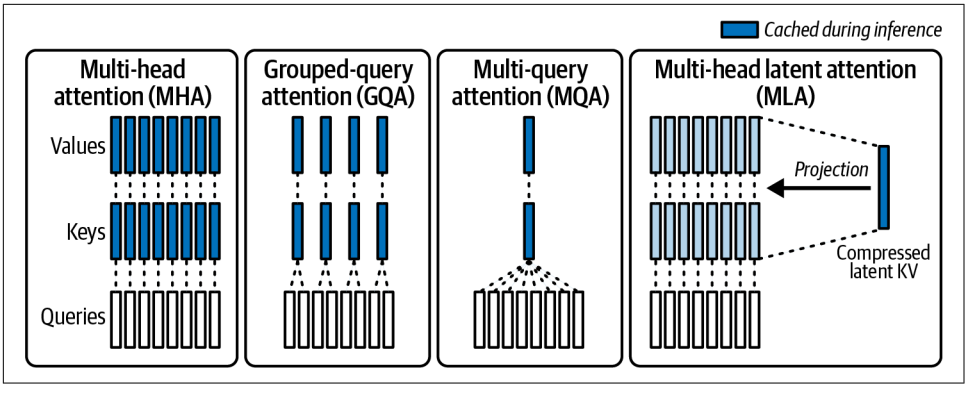

Figure 6-10. Comparing multi-head attention, multi-query attention, grouped-query attention, and multi-head latent attention

To improve performance, the third setup in the diagram, MQA, was introduced. In MQA, all queries share a single key and value head. For a 7B model and 70B model, there are usually 32 and 64 attention heads, respectively. In MHA, this means it will have 32 and 64 KVs, whereas in MQA, it will have only one. Thus, with MQA, KV cache size can be reduced by 32 and 64 times, which is a huge gain. However, MQA has been shown to degrade model accuracy significantly because of its aggressive setup.

To mitigate the accuracy issue of MQA, grouped-query attention (GQA) was introduced to balance performance and accuracy. GQA groups query heads into multiple groups, each with the same key and value. This setup has proven to be a good middle ground between MHA, which is not computationally efficient, and MQA, which takes too big an accuracy hit, and is now used in many model architectures.

Now let's quickly examine how to know if a model is using MHA, MQA, or GQA. If you look at the *config.json* file of Llama2, you can tell that it is using basic MHA, because the number of KV heads is equal to the number of attention heads:

```
"num_attention_heads": 32,
"num_hidden_layers": 32,
"num_key_value_heads": 32,
```

In the Llama3 config file, in contrast, `num_key_value_heads` is reduced to 8. This means it is using GQA, where each KV head is shared: 32 divided by 8 = 4 attention heads:

```
"num_attention_heads": 32,
"num_hidden_layers": 32,
"num_key_value_heads": 8,
```

Another notable advancement in KV cache efficiency is multi-head latent attention (MLA), introduced by DeepSeek, which is the rightmost setup in Figure 6-10. The

key difference between MLA and other variants is that, instead of simply reducing the KV counts, tt uses a clever way to compress them. DeepSeek's original paper claims to achieve a KV cache that "is equal to GQA with only 2.25 groups, but its performance is stronger than MHA."

All the above advancements, however, are happening at the model architecture level. This means the choice of model architecture or attention mechanism usually also depends on which model family or specific model works best for your specific use case. Thus, this is a holistic decision. These advancements in model architecture mark an important shift in the trend of model development, in that the focus is no longer solely on improving model quality but, increasingly, on productizing models. With more and more LLMs becoming an integral part of AI systems, the key to success is designing architectures that are not only powerful but also practical, scalable, and cost-efficient enough to serve in real-world systems.

Kernel Fusion and Custom Attention Kernels

Next, let's look at how to make attention computation even faster by using specialized GPU kernels. *Kernels* are small, specialized programs that execute on the GPU to perform computations such as matrix multiplications, softmax, and other crucial operations for LLMs and many other deep-learning models. Using properly optimized and specialized GPU kernels based on the given model architecture, hardware, and LLM workload can significantly improve your GPU utilization, inference speed, and throughput.

Kernel fusion

One key technique in kernel optimization, widely used in the general machine learning world as well as LLMs, is kernel fusion. *Kernel fusion* merges multiple individual operations (such as a multiplication operation and an addition operation) into a single one, to minimize the data-movement overhead between memory and compute. In this way, it reuses the data already in the register or shared memory, without needing the round trip of writing back to GPU global memory and loading again. Figure 6-11 illustrates how kernel fusion works by converting the shapes in one go, without writing and loading back and forth to and from memory in the middle step.

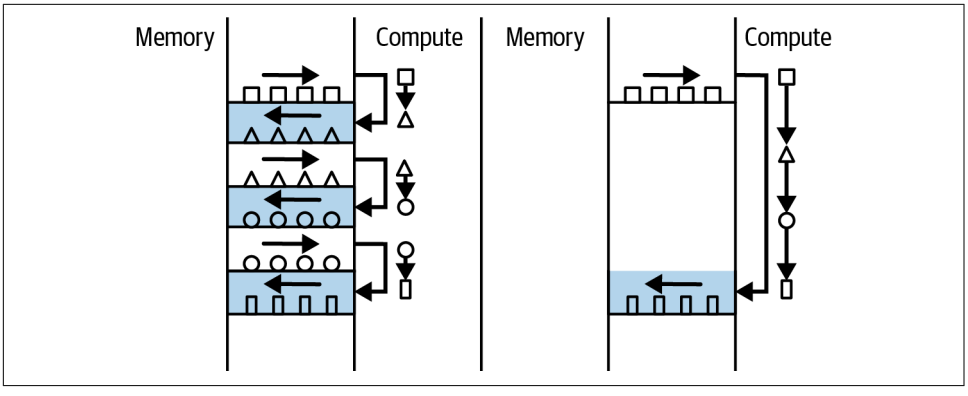

Figure 6-11. Comparing memory and compute interaction without kernel fusion (left) and with kernel fusion (right) (adapted from He, 2022)

FlashAttention

Next, let's look at one of the attention kernels in detail. FlashAttention represents a major advancement in high-performance attention mechanism computation. The core idea of FlashAttention is that, since attention is bottlenecked by GPU memory bandwidth during frequent reads and writes, making the algorithm hardware-aware (or memory I/O aware) to reduce the I/O combats that memory limitation.

Recall that in Chapter 5, we discussed the hierarchy of GPU memory bandwidth: the main GPU global *high bandwidth memory* (HBM) is the biggest within the GPU memory hierarchy, but it also has the slowest data movement. The goal of FlashAttention is to avoid materializing the big matrices in the slow GPU global memory, instead breaking up those large matrices into smaller ones. This is called *tiling* (or *blocking*), and it allows all the computation to happen in faster memory, like SRAM or even registers, without needing to utilize GPU global memory. Figure 6-12, from the original FlashAttention paper, shows how FlashAttention performs attention QKV matrix multiplication and conversions in tiles iteratively, so all the compute happens in GPU SRAM and only the final output is saved to HBM.

The FlashAttention kernel also fuses all attention operations, together with additional key ideas such as online softmax, to achieve one of the best attention kernels available. It is now used by many serving frameworks.

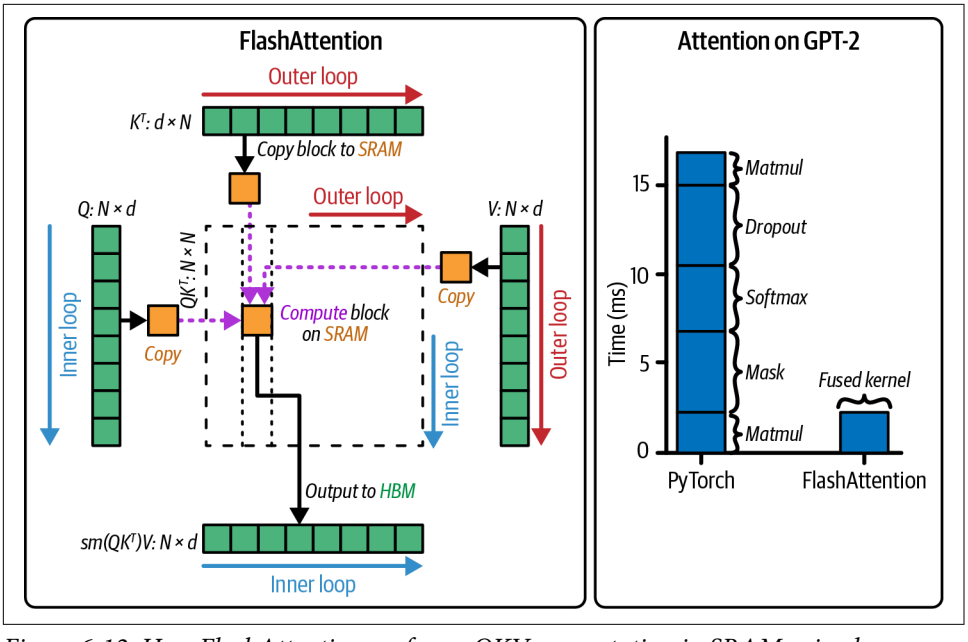

Figure 6-12. How FlashAttention performs QKV computation in SRAM using loops (left) and the performance gain of fused FlashAttention (right) (adapted from Dao et al., 2022)

Over time, FlashAttention 2 and 3 have improved on the original. They keep its core ideas but add techniques, such as overlapping General Matrix Multiply (GEMM) calculation with softmax calculation to improve GPU utilization—especially in newer-generation GPUs such as H100.

Kernel optimization is a big topic and an important research area that requires expertise in multiple areas such as GPU architecture, CUDA, performance profiling, and compilers, and is thus well outside the scope of any single chapter. What we want you to take away is this: from a practical and engineering perspective, when serving LLMs, it is important to leverage efficient kernels. Other optimized kernels include FlashInfer, xFormers, and Triton (not to be confused with Triton Inference Server, a totally different concept). They all enable high performance attention for fast model training and serving, exemplifying the power of GPU kernel optimization.

For example, with a few lines of code and environment variables, you can install a custom FlashInfer kernel and its dependencies to run in vLLM:

```
pip install vllm==0.8.5.post1
pip install flashinfer-python==0.2.2

export VLLM_ATTENTION_BACKEND=FLASHINFER
export VLLM_USE_FLASHINFER_SAMPLER=1
export VLLM_FLASHINFER_FORCE_TENSOR_CORES=1
```

It's similar in SGLang, with a tweak of a flag:

```
--attention-backend {flashinfer|fa3|triton|torch_native|FlashMLA}
```

Because of the complex and varied natures of different kernels, hardware, and LLM input/outputs, it is relatively hard to give clear-cut advice on which kernel to use. In our experience, finding the appropriate one usually requires some experiments. And many serving backends, such as vLLM and SGLang, have built-in logic to choose which attention kernels to use by default. For example, at the time of writing, SGLang defaults to FlashInfer for non-Hopper machines (like A100 and A40) and to FlashAttention3 for NVIDIA Hopper architecture GPUs (like H100, H200, and H20). It is OK in practice to start with the recommended one and pursue other optimization opportunities at first before experimenting with different kernels for additional performance gain.

PagedAttention

One last important way to improve attention is by efficiently managing the KV cache memory. During model serving, the system is constantly creating and storing new KV cache and evicting old cache. The KV cache size grows larger and larger as modern LLMs become capable of longer context lengths. Also, scheduling this large KV cache is never easy, because for LLM serving, the input length can vary by request and the output length is not predetermined. All these unique challenges can cause a lot of inefficiency in GPU memory, because traditional methods would preallocate memory space that usually isn't fully used, leading to significant memory fragmentation and low usage of real-world memory space.

An important advancement introduced to combat these problems is PagedAttention, along with the popular open source vLLM serving framework that builds on top of it. PagedAttention was inspired by the paging technique in operating systems that divides memory into fixed-size blocks called *pages* to make it easier to allocate scattered free spaces, and to enable long sequences without requiring contiguous memory. PagedAttention works in a similar way. It first partitions the KV cache into fixed-sized blocks. Then, using a lookup table, it maps query keys to specific blocks. This means the KV cache does not need to be stored in contiguous memory—the blocks can just be accessed individually when needed.

As you can see from Figure 6-13, the full prompt and completion combination is not stored in continuous space in actual physical memory, pictured on the right. They are stored in three different blocks, starting at block 7, then block 1, and now block 3. Each block contains a maximum of 4 tokens or words, with the last one containing only 2, as it is still actively generating. The *block table* is the lookup table used to find the physical block where the requested data is located.

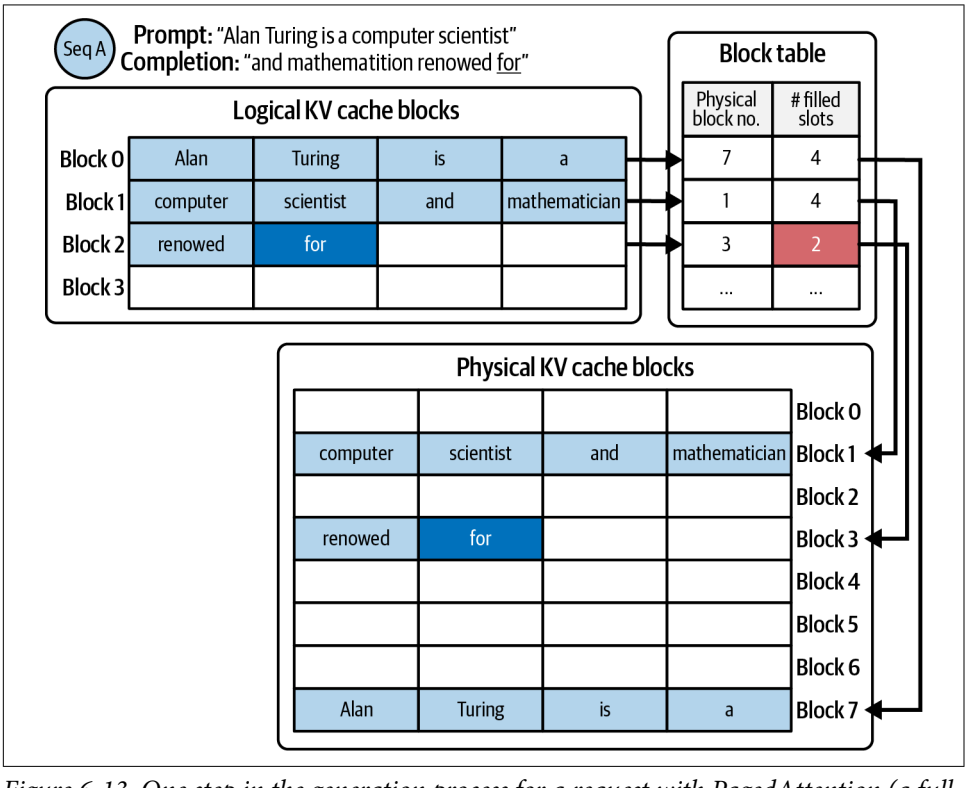

Figure 6-13. One step in the generation process for a request with PagedAttention (a full animation is available from the vLLM blog)

The original PagedAttention paper states that, without PagedAttention, "only 20.4%–38.2% of the KV cache memory is used to store the actual token states," but that with PagedAttention, it achieves "near-zero waste in KV cache memory."

Because of these great improvements in reducing memory fragmentation, PagedAttention and its variants have become a basic feature of LLM serving that is enabled for almost all usage, much like continuous batching. You can read much more about the implementation details of PagedAttention in vLLM's documentation.

Model Compression

LLMs have unlocked astonishing capabilities, but their sheer size causes so many problems in real-world production usage. Acquiring high-performance GPUs can be both costly and challenging. To bring these large and expensive models to consumers, we need to shrink them—smartly.

That's where *model compression* techniques come in. These are not crude hacks, but production strategies that reduce model size and computational load while preserving performance.

Model compression techniques normally fall into three categories:

Quantization
: *Quantization* reduces the precision of a model's parameter from higher-bit to lower-bit formats, to squeeze more parameters into memory and speed up matrix operations.

Distillation
: *Distillation* transfers knowledge from a large "teacher" model into a smaller and faster "student" that mimics its behavior.

Pruning
: *Pruning* surgically removes redundant weights or attention heads, revealing how much of the model's capacity is underused.

In this section, we will unpack each of these techniques, show their trade-offs, and explore practical ways to apply them.

Among the three major model compression techniques, quantization stands out in terms of practicality. It is fast, effective, and typically requires little to no modification of the model training pipeline. These advantages make quantization the go-to choice for compressing and accelerating LLMs in production environments. In fact, quantized models are more widely used in production than you might expect—particularly in scenarios that demand low latency and high throughput, or where models need to run on resource-constrained edge devices.

Given quantization's impact and prevalence, we'll spend more time diving into it in this section than the other two techniques.

Quantization

Quantization is the process of reducing the precision of a model's parameters, such as weights, activations, and KV cache, from high-precision floating-point number formats like 32-bit (FP32) and 16-bit (FP16 or BF16) to low-bit representation, such as 8-bit (FP8 or INT8) and 4-bit (FP4 or INT4). With quantization, you're essentially trading lower accuracy in the model data for better serving performance.

Quantization error

During quantization, you are reducing the precision of the model parameter's numerical value from high to low. This introduces two types of errors—rounding errors and clamping errors—that can degrade the model's inference accuracy.

Rounding errors happen when a number (like FP32) can't be represented exactly in a given numerical format (like INT8), so the system rounds it to the nearest representable value. The difference between the original value and the converted value is the rounding error. For example, let's say we have a FP32 value of 7.6. When we convert it to INT8, since INT8 can't store the decimal, 7.6 gets rounded to the nearest integer: 8. The rounding error here is thus 8 – 7.6, or 0.4.

Clamping errors happen when a number is too large or too small to fit in the numerical format's range. In these cases, you can "clamp" the number to a maximum (or minimum) representable value. This happens because formats like FP8 and FP16 have limited exponent ranges. For example, suppose FP8 can only represent values between –448 and +448. If your number is 1,000, it's too big to present in FP8. So, during FP32 to FP8 compression, you would need to clamp 1,000 to 448 (the maximum allowed value in FP8).

Because clamping can introduce severe distortion—for example, turning 4,096 into 448 in FP8—modern quantization techniques avoid hard clamping when possible. Instead, it's better to use scaling strategies to better preserve the relative magnitude of values within the target range, even if some rounding error is introduced.

The idea is to apply a scaling factor during quantization to compress the value range of the original data, so that more values fall within the representable range of the lower-precision format (like INT8 or FP8) rather than being abruptly clipped at the extremes. The common scaling strategies we have used are symmetric scaling and asymmetric scaling, as covered on Maarten Grootendorst's blog "A Visual Guide to Quantization."

Now that you have a general understanding of quantization, let's take a closer look at the different data formats used to store LLM parameters.

Storing numbers

Floating-point (FP) format is a standardized way for computers to represent real numbers, such as 3.14, –42.7, or 1e9. The general structure of a floating-point number is:

Total bits = 1 sign bit + mantissa bits + exponent bits

This represents a number in three parts:

- The *sign bit* (1 bit) indicates whether the number is positive or negative.
- The *mantissa* (or *significand*) controls the precision (level of detail) of the number's representation.
- The *exponent* controls scale, determining how large or small the number can get.

There are different floating-point formats, but the most common are FP32, FP16, and BF16. Table 6-1 provides a detailed breakdown.

Table 6-1. Comparing common floating-point number formats and their bit allocations

Precision format	Total bits	Sign bit	Exponent	Mantissa	Approx. range level
FP32	32	1	8	23	$\pm 10^{38}$
FP16	16	1	5	10	$\pm 10^{-5}$
BF16	16	1	8	7	$\pm 10^{38}$

FP32 is considered full precision and uses a total of 32 bits: 1 sign bit, 8 bits for exponent, and 23 for mantissa. Its wide value ranges and good precision, as shown in Table 6-1, make FP32 the standard format for training deep-learning models.

The FP16 and BF16 formats are both half the size of FP32, at 16 bits. The trade-off made between FP16 and BF16 is that using more bits for exponent or for mantissa results in getting better precision or a better value range, respectively.

Compared to FP16, BF16 allocates more bits to the exponent (8 bits, versus 5 in FP16), which trades off precision (fewer mantissa bits) for a significantly wider numerical range. Interestingly, both BF16 and FP32 use 8 bits for the exponent, meaning they share the same value range. This makes conversion from FP32 to BF16 more straightforward than from FP32 to FP16—there's no risk of clamping due to range limits, only a loss in precision due to the reduced mantissa. This design makes BF16 particularly well-suited for deep learning training workloads, where dynamic range is more critical than fine-grained precision, while also reducing memory and compute overhead. Figure 6-14 is a visual comparison of the FP32, BF16, and FP16 formats.

In current LLM serving schemes nowadays, FP32 precision is less prevalent, and most model checkpoints are provided either in FP16 or BF16.

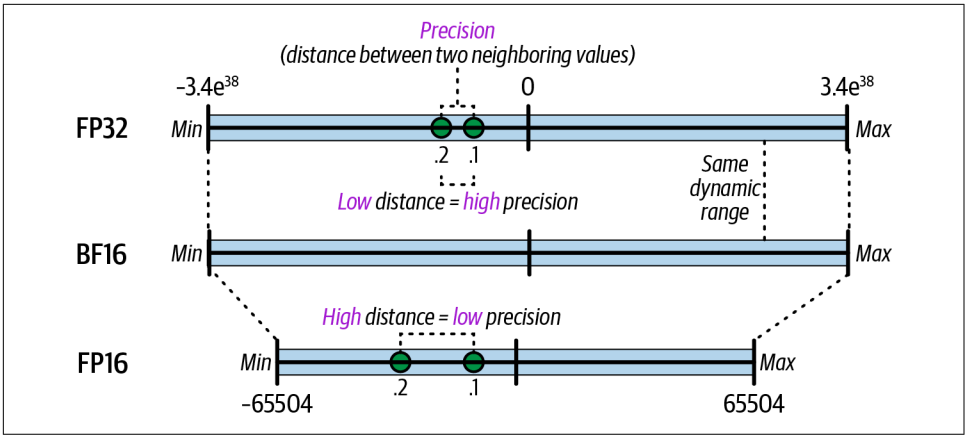

Figure 6-14. Visualization of FP32, BF16, and FP16 precision and range (adapted from Grootendorst, 2024)

Now that you've seen the full-size data format, let's examine the quantized number formats. There are two types of quantized number representation: *integer-based representation,* such as INT8 and INT4, and *floating-point-based representation,* such as FP8 and FP4, which have become more popular even while we've been writing this book.

The key difference between these two formats is how they distribute data points. In the integer-based format, the data distribution is completely uniform, meaning the data points are evenly spaced across the whole value range and the number precision does not change. In contrast, floating-point has nonuniform distribution: that is, there are more data points around zero and less data points at the extremely large or small numbers.

The difference in data density between the FP and integer-based formats is because FP number values are logarithmic. An FP number's value (in FP8 format) is:

$$Value = (-1)^{sign} \times (1 + mantissa) \times 2^{exponent - bias}$$

As you increase the exponent, the distance between representable numbers also increases. For example, between 1.0 and 2.0, you can represent many values (like 1.01, 1.001, and so on), but between 1,000,000 and 2,000,000, the smallest step size becomes much larger, like 128 or more (see Figure 6-15).

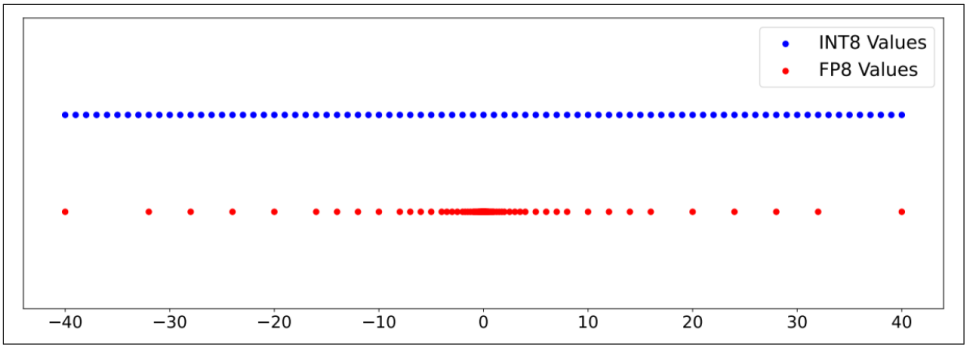

Figure 6-15. Data point distribution in INT8 and FP8 formats

Depending on the data distribution of a given model's parameters (for example, model weights and activation values), you may want to choose one format over the other to retain the best accuracy.

How does quantization help in model serving?

It is very important to understand why and how quantization helps increase model serving throughput and reduce latency, so that you can choose the best quantization method for your use case.

The first reason is data size. Whether you need 8 or 16 bits directly impacts the model size, on disk and in GPU memory. Recall (from Table 5-6 in Chapter 5) that FP16 precision corresponds to 2 bytes (1 bytes = 8 bits) per parameter. Here's how you would calculate the data size of a 7-billion-parameter model with FP16 precision:

$$\sim 7 \text{ billion parameters} \times 2 \text{ bytes/parameter} = 14 \text{ billion bytes} = 14 \text{ GB}$$

However, quantizing this model to INT8 immediately cuts its size in half, to 7 GB. This is a huge win when you don't have a big enough GPU memory to fit the unquantized model.

Similarly, when serving a large model, quantizing it can mean fitting it in one machine or node instead of needing multiple nodes and dealing with cross-node communications. The reduced memory footprint also frees up space for the KV cache, allowing the system to handle more concurrent requests and boosting its overall throughput.

You also learned in Chapter 5 that GPU memory bandwidth is the key to LLM performance optimization, especially in the decode phase, since data movement is the bottleneck. Quantizing the model and making it smaller directly reduces the data movement needed, which greatly helps to reduce model inference latency.

Lastly, lower precision can also mean faster computation. You looked at a similar table in Chapter 5, Table 5-2, when we compared GPU hardware. Now look closely at Table 6-2, which compares FLOPS across different precisions. It is apparent that when we reduce the bits by half, from 16 bits to 8 bits, we typically can double our FLOPS; the same goes for quantizing weights to 4 bits.

Table 6-2. H100 FLOPS across different precisions

	H100 SXM
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core	989 teraFLOPS
BFLOAT16 Tensor Core	1979 teraFLOPS
FP16 Tensor Core	1979 teraFLOPS
FP8 Tensor Core	3958 teraFLOPS
INT8 Tensor Core	3958 TOPS

In short, quantization can help reduce model size, reduce data movement during computation, and speed up computation.

Weight-only versus weight-and-activation quantization strategies

Quantizing on the model weight only and quantizing on both the weight and activation are two of the most common quantization strategies. You might have seen some quantization notations like W4A16 and W8A8: this type of notation is used to describe the bit widths used for a model's weights (represented by *W*) and activation parameters (*A*), which correspond to the model's middle-step inputs and outputs. For example, W4A16 means 4-bit model weights and 16-bit activations, and W8A8 means 8-bit model weights and 8-bit activations

Weight-only quantization means that the model weights are quantized but the activation parameters are not. This reduces the model size but, during execution, the lower-bit values are dequantized back to high-bit formats. Thus you benefit from the reduction in model size and data movement, but do not achieve faster computation. In fact, because you need to dequantize the weight before computation, it adds a little overhead.

To avoid dequantization in model inference, you can use mixed-precision kernels, such as the Marlin kernel for Ampere-generation NVIDIA GPUs such as A100, and the Machete kernel for Hopper-generation NVIDIA GPUs, such as H100. These kernels can efficiently handle matrix multiplications by fusing the quantized weight access. They can perform multiplication between, say, an INT4 matrix and an FP16 matrix in one pass without dequantization. In practice, many people turn these

kernels on by default when serving weight-only quantized LLMs in LLM serving engines.

Weight and activation quantization not only reduces model size and data movement; because it quantifies activation, it also helps achieve higher FLOPS with a compute-bound workload. However, activating it is more complex than with weights-only because complexity depends on both model weight and model input, which changes all the time. This means that, when quantizing activation, you need to choose when to do the scaling. You have two options. With *dynamic scaling*, you're basically calculating the scaling factor on the fly as you do the inference. In *static scaling*, the scaling factor is estimated before the model is deployed and precalculated using a calibration dataset. The trade-off is that, while dynamic scaling gets better accuracy, it is less performant than static scaling because it calculates the scaling factor during the actual inference.

For weight-only quantization, the most commonly used at scale in production is W4A16, using the GPTQ or AWQ quantization methods. For weight-and-activation quantization, the most commonly used in production at the time of writing is W8A8 (which was previously in INT8 format and recently moved to FP8 format). Table 6-3 compares the benefits of adopting quantization on W4A16 versus on W8A8.

Table 6-3. Comparing W4A16 and W8A8 quantization

	W4A16	W8A8
Model size reduction/data movement	75% (original size/4)	50% (original size/2)
Compute FLOPS	No change	2x
Prefill phase (compute-bound)	No change	Improved
Decode phase (memory bandwidth-bound)	Improved (better at low batch)	Improved (better at high batch)
Best for	Long generation, latency sensitive, low batch	Long context, high throughput, high batch

W4A16 reduces the model footprint twice as much as W8A8, making it occupy less GPU memory and require less GPU memory bandwidth, so it greatly helps with the decode phase. Comparatively, W8A8 excels in compute-bound workloads such as long-context prefill phases and high-batch, high-throughput scenarios where the decode phase is batched and pushed towards being compute bound.

How to Choose a Quantization Strategy

In real-world production deployment, if the model is large and requires 4x model size compression to fit inside a single GPU to avoid cross-GPU communication, or GPUs within a single machine to avoid cross-node communication, W4A16 is the way to go. On the other hand, the benefit of quantizing activation in more efficient compute usually outweighs the memory saving in high batch, moving the bottleneck of the model execution from memory bandwidth-bound to compute-bound. For production workloads, if we can achieve a certain latency SLA with W8A8, not needing W4A16, we try to push for higher effective batch and high throughput per model instance to reduce cost.

Now, let's revisit the data format. In the past, W8A8 quantization was mostly done using the INT8 format, where both weights and activations were clamped to the fixed ±127 integer range for simplicity and hardware compatibility. Recently, a lot more models have been deployed to production in FP8 variants (such as E4M3 and E5M2) instead. In a 2022 paper, NVIDIA introduced the FP8E4M3 format and showed how it can improve serving performance while maintaining minimal accuracy loss compared to FP16, without needing calibration to maintain model accuracy (like INT8 does).

Among the two variants of FP8, E4M3 is more commonly used for inference because it retains more precision than the E5M2 variant, despite E4M3 having smaller dynamic range and thus a potential need for scaling. Table 6-4 compares the two.

Table 6-4. Floating-point precision formats and their bit allocations

Precision format	Total bits	Sign bit	Exponent	Mantissa
FP8 (E4M3)	8	1	4	3
FP8 (E5M2)	8	1	5	2

FP8 Is Not Always Applicable

Another thing to consider is hardware compatibility: not all GPUs or accelerators support FP8 format. Of the NVIDIA GPUs, only Hopper and Blackwell support FP8. This means that if you have only A100s or older GPUs, running FP8 might not achieve the expected full performance gain.

Because this is a more engineering-focused book, we will not go deep into the quantization process. For more information, we recommend Grotendorst's "A Visual Guide to Quantization".

Hands-on quantization

In this section, let's try some examples to see how to apply quantization in practice.

In the Google Colab notebook, you can find the code that hosts the Qwen/Qwen2.5-7B-Instruct model in three different variations: the original, GPTQ W4A16 quantized, and W8A8 quantized in a vLLM server. For each model, we'll show you how to conduct benchmark tests with different concurrencies. You'll record their TTFT, time per output token (TPOT)/ITL, and throughput to better understand the performance gain and trade-offs between them.

Find the quantized model. In the notebook, we execute both the original model and quantized versions of the models as a standalone server. To execute a model, you'll first need to either save it to a local or cloud location or directly reference a model on Hugging Face. In many cases, the popular foundation models and their quantized versions can be found directly in Hugging Face so you might not have to quantize the model yourself. For example, Hugging Face provides all the quantized variants of the model uploaded already, on the right-hand side, as shown in Figure 6-16.

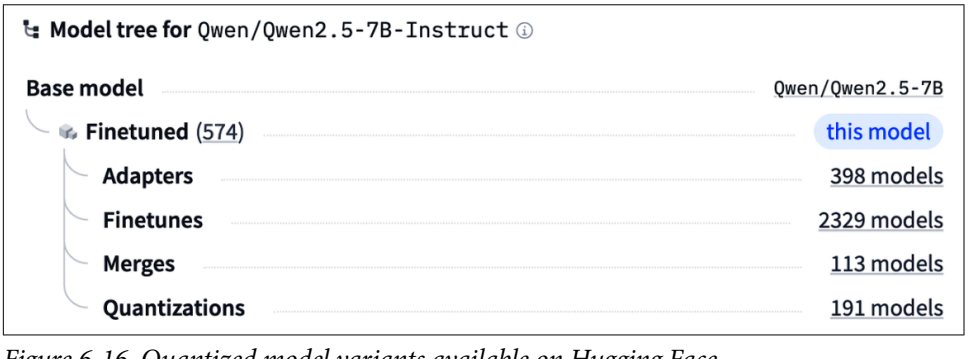

Figure 6-16. Quantized model variants available on Hugging Face

Quantize a model yourself. In this case, you can simply pass the model ID, `Qwen/Qwen2.5-7B-Instruct`, as an argument to the server startup command. You can also replace the model ID with a model path if needed:

```
hf_model_id = "Qwen/Qwen2.5-7B-Instruct"
!vllm serve {hf_model_id}
```

If you want to use a different calibration set to quantize a foundation model, or quantize a custom fine-tuned model yourself, there are many help libraries, such as GPTQModel, AutoAWQ, and LLMCompressor.

Here's a code snippet for GPTQ quantization:

```
from transformers import AutoModelForCausalLM, AutoTokenizer, GPTQConfig

tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-7B-Instruct")
dataset = [
    "Gptq is an easy-to-use model quantization library with user-friendly APIs, "
    "based on the GPTQ algorithm."
]  # calibration dataset
gptq_config = GPTQConfig(bits=4, dataset=dataset, tokenizer=tokenizer)

quantized_model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2.5-7B-Instruct",
    device_map="auto",
    quantization_config=gptq_config
)
```

Running benchmarks. Once the vLLM server is up, you can start sending it requests and observe how models quantized using different methods perform. To run the benchmark, use the benchmark script provided inside vLLM's repo.

Performance analysis. After running the benchmark scripts for all three model variants, we gathered the performance figures and plotted them, as shown in Figures 6-17, 6-18, and 6-19.

You can see that in low-concurrency settings, GPTQ W4A16 provides the best performance, with about a 300% performance gain in latency and throughput over the original model. FP8 W8A8 also provides a decent gain at 150%. In low-batch settings, the bottleneck is on GPU memory bandwidth. GPTQ reduces the weight to INT4, which is four times smaller, and that really shines here. For use cases where you need the best latency, such as chatbots or AI agent flows (where multiple calls to the model are needed and the aggregated latency reduction is crucial), W4A16 weight-only quantization is the way to go.

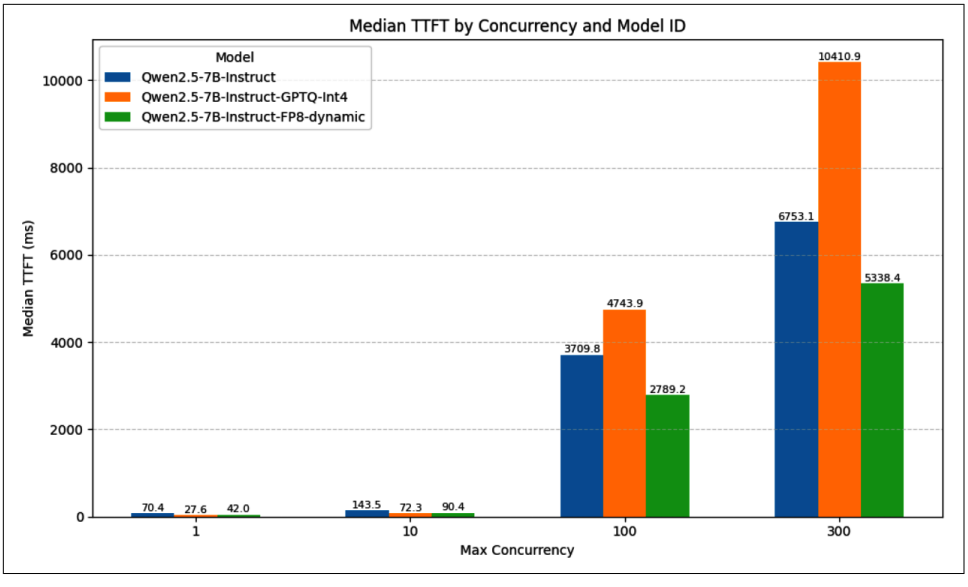

Figure 6-17. Comparing TTFT between the original model and models quantized with GPTQ-Int4 (W4A16 weight-only quantization) and FP8 (W8A8 weight-and-activation quantization) on Qwen2.5-7B-instruct model under different request concurrency

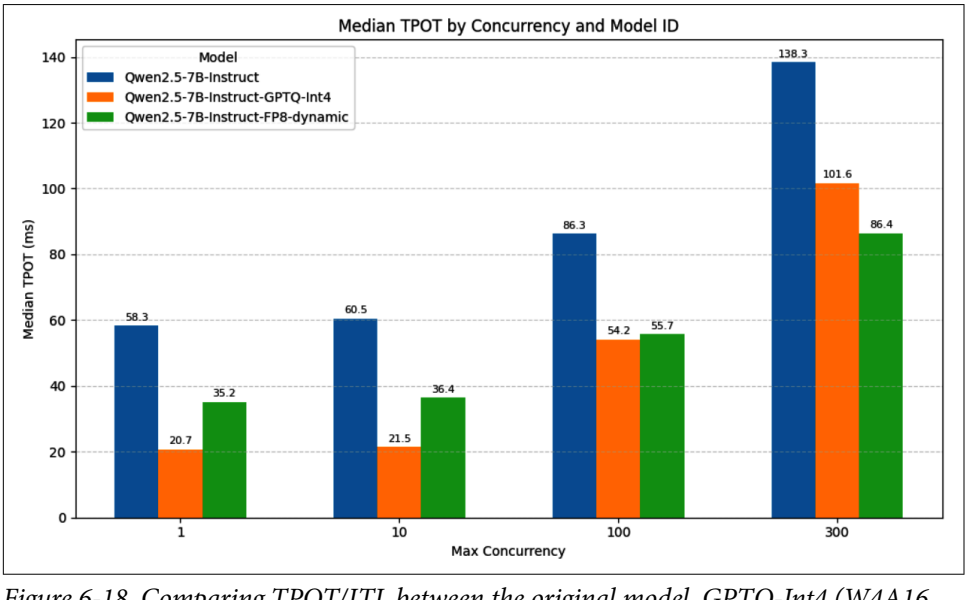

Figure 6-18. Comparing TPOT/ITL between the original model, GPTQ-Int4 (W4A16 weight-only quantization) and FP8 (W8A8 weight-and-activation quantization) on Qwen2.5-7B-instruct model under different request concurrency

Next, let's see what happens as concurrency goes up along the x-axis from the left to the right side of each chart, in high-batch processing. GPTQ W4A16 shows its weakness here because, as a weight-only quantization method, its computation is still in 16-bit. The TTFT increase (Figure 6-17) is most apparent with GPTQ W4A16, which is even slower than the original model. This is because it does not save anything in computation and incurs dequantization overhead. FP8 W8A8, on the other hand, performs much better now because the activations are also quantized to FP8, resulting in faster computation. If W8A8 latency is already good enough to satisfy SLA requirements for your use case, this is the time to push more batches, so that each model instance can serve more requests in parallel to save cost.

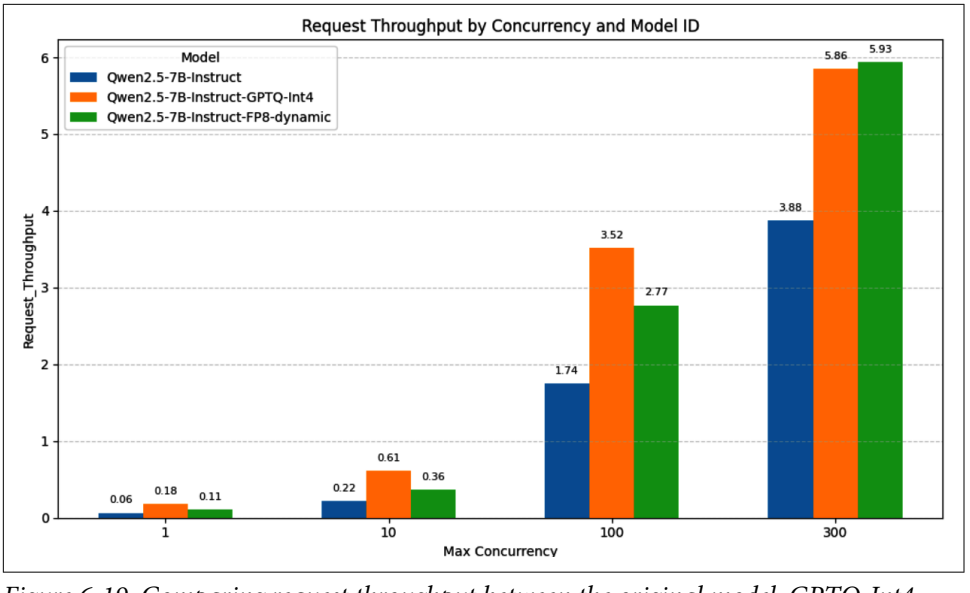

Figure 6-19. Comparing request throughput between the original model, GPTQ-Int4 (W4A16 weight-only quantization), and FP8 (W8A8 weight-and-activation quantization) on Qwen2.5-7B-instruct model under different request concurrency

Other quantization methods

So far, when we've talked about quantizing the weights and activations, we are mostly focusing on the feedforward layers (FFN), not the attention layers. Even though feedforward layers comprise the majority of the parameters and computation, the attention mechanism and KV cache quantization are also applied in certain cases.

KV cache quantization and attention quantization. The first step is quantizing the KV cache. In both weight-only and weight-and-activation quantization, the KV cache is usually left in a high-precision format. But KV cache can take up a significant amount of GPU memory. Quantizing the KV cache can free up more GPU memory. And

with more GPU memory, you can increase the batch size to gain higher throughput. Quantizing the KV cache can also benefit techniques such as prefix caching, allowing more prefixes to be cached without recalculating or reloading. (We will discuss prefix caching in detail later in this chapter.)

However, quantizing KV cache usually will not significantly reduce latency. This is because, if the attention calculation is still using a high-precision format, quantizing the KV cache only makes it smaller. That doesn't mean you can gain more FLOPS during the calculation. During the attention calculation, similar to weight-only quantization, quantized KV cache also needs to be dequantized back. Thus, to fully reap the benefit of a quantized KV cache, you need to use it in conjunction with quantized attention kernels.

Overall, we usually still start with weight-only or weight-and-activation quantization. For example, we start with FP8 with both weight and activation. Then to tackle long context in limited GPU memory or to gain more throughput for decoding a heavy workload, we add FP8 KV cache quantization and enable an FP8 attention kernel when needed.

GGUF quantization. GPT-Generated Unified Format (GGUF) is a very different type of quantization. GGUF is popular among users seeking local, portable, and low-resource deployments because it focuses on running LLMs on CPU and/or Apple Silicon (metal) instead of GPU, but offloads some parts to GPU when needed, if available. It also provides a lot of different quantization for levels and formats to meet your needs when high-end GPUs are not available. For more information on GGUF see the library on GitHub, along with how to run models locally with Llama.cpp with GGUF from models on Hugging Face.

Accuracy trade-offs

So far we have examined many different quantization techniques and their trade-offs. However, the biggest trade-off when applying quantization is between model accuracy and serving performance (throughput and latency). If a quantized model can't maintain the accuracy of the original model to sustain enough quality, all the performance gain cannot be materialized because the inferior product can never be deployed to production. Luckily, there's been a lot of testing and research on maintaining model accuracy. Nowadays, methods such as GPTQ W4A16, AWQ, and FP8 quantization on both weights and activation have all shown minimal loss in real-world accuracy metrics, as shown in Figure 6-20. On the other hand, the serving performance gain from quantization sometimes enables you to run a quantized version of a bigger model. For example, you could deploy a 12-billion-parameter model in FP8 over an 8-billion-parameter model in FP16 to achieve comparable latency, higher throughput, and better model accuracy.

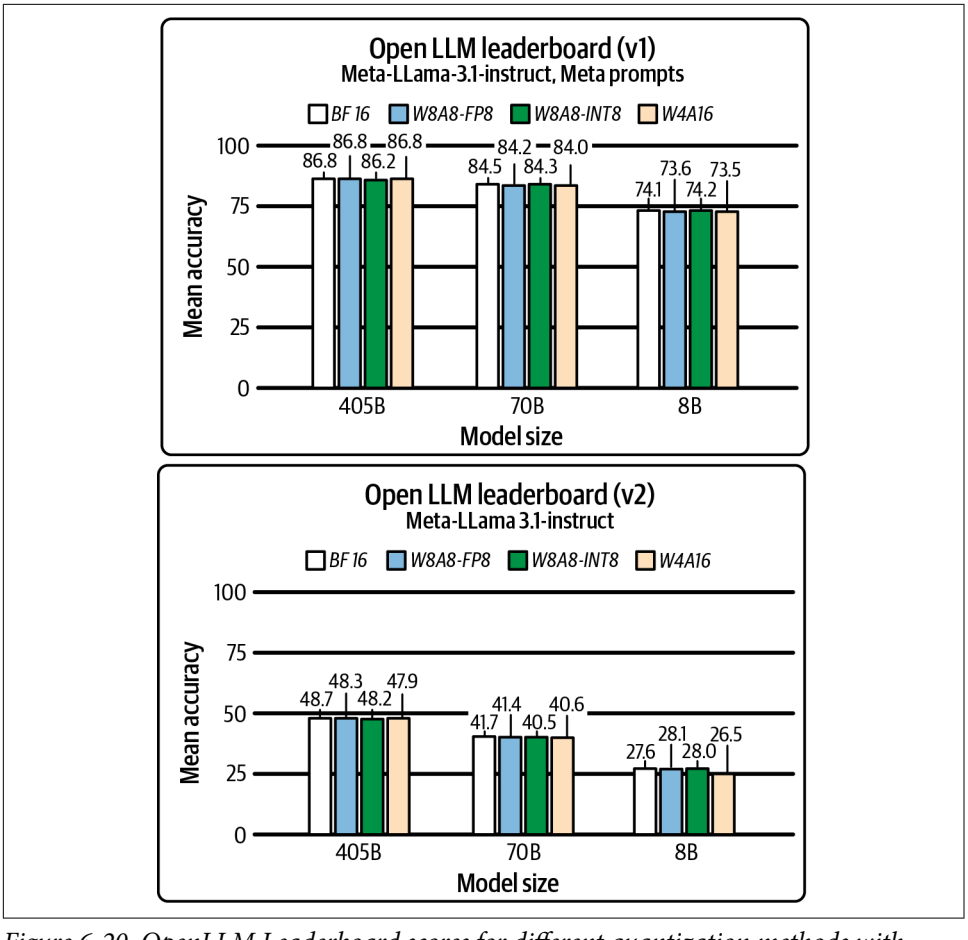

Figure 6-20. OpenLLM Leaderboard scores for different quantization methods with different model sizes, by Neural Magic (adapted from Kurtić et al., 2024)

This research has also shown some patterns in how different quantization methods affect models of different sizes and context lengths, which can guide you in finding the best quantization level and method to retain satisfactory accuracy. For example, you could try to push the boundary with more aggressive weight-only quantization and long context serving; the larger the model is, the more sensitive its accuracy will be to model weight and KV cache quantization. KV cache quantization is relatively less intrusive in terms of accuracy but doesn't yield a significant performance gain. If you're struggling with KV cache space and ITL, it is still a good method, but less of a priority.

After quantization, you can evaluate the model's accuracy to make sure it is still satisfactory. A commonly used tool is LM Eval, which supports a wide variety of setups.

Here is a code snippet for accuracy evaluation:

```
lm_eval --model {hf|vllm|sglang} \
    --model_args Qwen/Qwen2-7B-Instruct \
    --tasks gsm8k_cot \
    --device cuda:0 \
    --batch_size auto
```

Current research is moving towards even lower bits such as FP4, FP6, and W4A8 quantization with newer Blackwell generation NVIDIA GPUs. So far, it has been quite hard to maintain good model accuracy, but thanks to per-tensor and per-channel scaling, outlier-aware clipping, and newer techniques all working together to minimize quantization error, expect to see these lower-bit options used in production in the near future.

Quantization-aware training

One powerful but complex way of mitigating accuracy issues in quantization especially with very low bits is performing *quantization-aware training* (QAT). All of the quantization techniques we've described so far focus on *post-training quantization* (PTQ), which doesn't require access to the training pipeline at all. This enables you to quantize the foundation and custom-trained models you have access to but did not necessarily train yourself. This is why, in practice, we've found that post-training quantization is a lot more popular than quantization-aware training because it's easier to use and still decently accurate.

QAT performs fake quantization during the training phase or often, more specifically, the fine-tuning and alignment phase, to better preserve accuracy. QAT has superior accuracy because it allows the model to adjust its weights during training to compensate for precision loss. This makes it often the only reliable choice for aggressive compression like 4-bit level quantization where simple post-training rounding would cause the model to fail. Table 6-5 lays out a comparison between PTQ and QAT.

Table 6-5. Comparison of post-training quantization and quantization-aware training

	Post-training quantization	Quantization-aware training
When applied	Completely after training on static weights	During training/fine-tuning with quantization effects simulated
Difficulty	Much lower, conversion and some calibration	Much higher, extra compute for training pipeline
Accuracy on higher bits (8 bits and up)	Good	Good

	Post-training quantization	Quantization-aware training
Accuracy on lower bits (4 bits and below)	Usually not acceptable	Better and acceptable
Flexibility	Easily done, custom tuned and deployed to different hardware	Tied to the quantization scheme causing it to be less flexible for further fine-tuning and deployment on different hardware

A good real-world example of this is the open source GPT-OSS family of models released by OpenAI. It uses FP4 E2M1 (further read for the MXFP4 format) QAT techniques to aggressively shrink the model size.

Based on the original document from OpenAI, "The gpt-oss-120b model achieves near-parity with OpenAI o4-mini on core reasoning benchmarks, while running efficiently on a single 80 GB GPU. The gpt-oss-20b model delivers similar results to OpenAI o3-mini on common benchmarks and can run on edge devices with just 16 GB of memory."

In FP4 format, each parameter is now only 0.5 bytes/parameter. Let's do our simple model size estimate on the gpt-oss-120b model, which has 117 billion parameters, and the 20b model, which has 21 billion parameters, as OpenAI published:

~117 billion parameters × 0.5 bytes/parameter = 58.5 billion bytes

= 58.5 GB < 80 GB

~21 billion parameters × 0.5 bytes/parameter = 10.5 billion bytes

= 10.5 GB < 16 GB

In fact, not all of the parameters of the GPT-OSS models are quantized to FP4. The quantization is only done on the mixture of experts (MoE) layers, which we will talk about later in Chapter 7. But because over 90% of the parameters are in MoE layers, the estimation is still valid.

The GPT-OSS family of models has gained a lot of popularity due to its performance in accuracy and inference speed. With the model shipped in a ready-to-run quantized model format, it can be deployed without a quantization algorithm and precision selection, saving a lot of deployment iterations. However, the main trade-off here is that for these gains to be realized, your hardware and inference stack need to be able to execute the MXFP4 format it ships with natively. FP4 formats are not equally supported across all GPUs so the performance will be highly dependent on the GPUs available. Blackwell (B200, for example) introduces dedicated 4-bit floating-point support such as NVFP4 (E2M1) and related block-scaled FP4 formats like MXFP4, while Hopper (H100/H200, for example) is architected around FP8 Tensor Cores (E4M3/E5M2) rather than native FP4. Thus, if Blackwell GPUs are available for

deployment, GPT-OSS model deployment is the best case. Hopper can still be very usable given the custom software-level support in mixed precision kernels. But if you only have Ampere (A100/A10, for example) or even older, you may be better off choosing a different model architecture as a whole, such as Qwen, and performing your own quantization.

Distillation

Now let's look into another technique in model compression, one that we believe has the biggest potential of the three to improve latency and throughput: model distillation. Model distillation is unique compared to quantization and pruning (discussed in the next section) in that, instead of trying to reduce the size of the original model, it basically trains a new small model. It tries to transfer the knowledge encoded in the large original "teacher" model into a smaller "student" model. In that process, the student model mimics the teacher model.

Figure 6-21 shows how a teacher model can generate outputs that can be used to train a much smaller student model. The outputs from the teacher model aren't necessarily limited to just the output token (which is usually called the *hard label*), but also can include logits for the probability distribution of the prediction and losses. Thus, to create a distilled student model, you need full access to the original teacher model, not just API calls to get the output tokens.

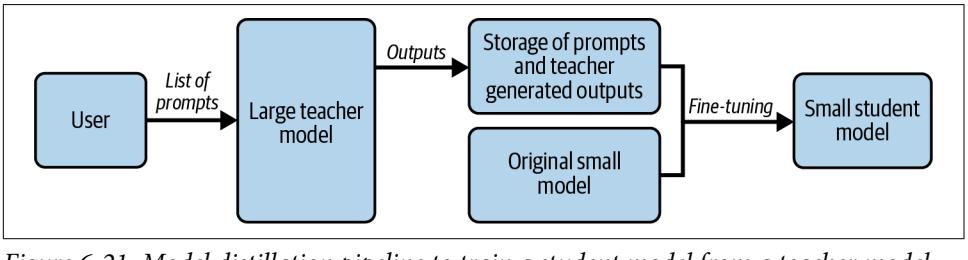

Figure 6-21. Model distillation pipeline to train a student model from a teacher model

Let's take DeepSeek's distilled models as an example. The original DeepSeek R1 model has 671 billion parameters and an MoE architecture (we will discuss MoE models in Chapter 7). DeepSeek has also released multiple models distilled from the Llama and Qwen families of open source dense models, ranging from 1.5 billion to 70 billion parameters. From a serving perspective, this reduces model size by 10 times or more and can mean huge improvements in serving latency and throughput. Table 6-6 compares the original with a 70-billion-parameter distilled model on the benchmarks.

Table 6-6. Comparing evaluation benchmarks between the original DeepSeek R1 and its distilled 70-billion-parameter model (source: DeepSeek)

Benchmark	DeepSeek-R1-671B	DeepSeek-R1-Distill-Llama-70B
MATH-500 pass@1	97.3	94.5
GPQA Diamond pass@1	71.5	65.2
LiveCodeBench (Pass@1)	65.9	57.5

Table 6-7 shows the pros and cons of quantization and distillation. If you're trying to figure out whether to use a distilled model or distill your own model to improve serving performance, our general guidance is to start with an easy, low-cost solution first. Let's say you are considering deploying either DeepSeek-R1-671B orDeepSeek-R1-Distill-Llama-70B. If the distilled model is already available, you can start by evaluating it to see if it meets your accuracy benchmark. If it is good enough, go with the distilled model. You can perform quantization on the distilled model to further improve serving performance and save cost. However, if you do not have a distilled model ready to use, which is likely to happen most of the time in real life, quantization should be your first step, given the high cost of performing distillation and the bigger accuracy loss.

Table 6-7. Comparing quantization and distillation

	Quantization	Distillation
Accuracy drop	Low (usually 3% or less)	Much higher than quantization
Speed gain	1.5x to 3x	Much more, with a trade-off in accuracy
Ease of use	Very easy for post-training quantization; you only need the original model weight.	Harder if the distilled model is not readily available; training costs can be as much as 10% of the original model cost. Usually done by researchers who trained the original model.

Pruning

The last model compression idea we'll look at, model pruning, is also the least popular, since (as we write this in mid-2025) more work and research are still needed to make it work for production. The core idea of *model pruning* is that models are usually overparameterized; pruning their redundancies can achieve better compression and thus improve serving performance.

Pruning is usually divided into structured and unstructured: *structured pruning* removes sections from the model entirely, while *unstructured pruning* removes individual weights with more flexibility. Notably, in the middle lies *semi-structured sparsity*, also called *2:4 structured sparsity,* where 2 out of every 4 elements are pruned. One semi-structured sparsity model is Sparse Llama 3.1 by Neural Magic, which claims to have achieved 98% accuracy recovery, 30% higher throughput, and 20% lower latency from sparsity alone with vLLM.

In Figure 6-22, for every group of four consecutive values in the original matrix on the left, two values are zeroed out (shown in white). This forms a sparsity rate of 2 (values shown as shaded green):4 (original total is four, with two now zeroed out in white), or 50%. In Figure 6-22, the matrix post compression is shown on the right still forming a dense matrix for computation. NVIDIA's GPU architectures, such as Ampere and Hopper, have sparse Tensor Cores that accelerate computation for this type of structured sparsity. The 50% sparsity can directly yield double the speed in matrix multiplication, which is a significant bump in performance.

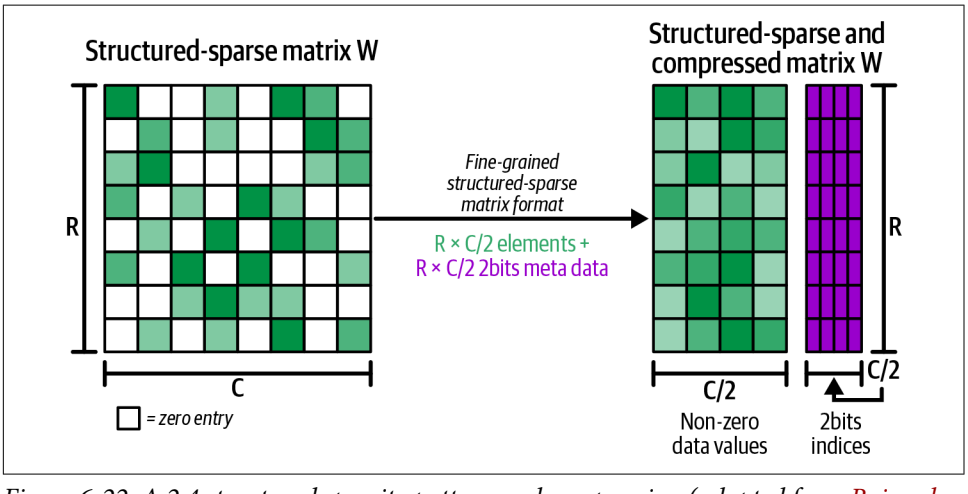

Figure 6-22. A 2:4 structured sparsity pattern and compression (adapted from Bai and Li, 2023)

Next, we'll finish up the chapter with a look at prefix caching.

Prefix Caching

In software engineering, caching is a very common technique for storing frequently accessed data in fast, possibly temporary storage to reduce latency and improve overall performance. In machine learning, caching the model prediction outputs of the same requests is very common in production, done on the client side, the server side, or both. The client side can use caching to avoid sending unnecessary duplicate requests, while the server side can use it to return the cached output without executing the model, to reduce computation.

An example of response caching on the server side is in Triton Inference Server, where each inference request (including model name, model version, and model inputs like tensor name, shape, datatype, and data) are hashed and cached along with the model output as key-value pairs. When a new request comes in, if the hash is found, the inference output is extracted directly from the cache.

The main reason request caching works is that storage is a lot cheaper than compute, and thus accessing storage is a lot faster than recomputing the result. However, while a general request-caching solution will help ML inference, the gain in LLM serving is usually small because LLMs take free-form human-written text as input. People can ask the same question in so many different ways that the cache hit rate of the naive request hashing can be very low.

A widely used, proven, powerful caching technique in LLM serving is *prefix caching*. Instead of matching the entire prompt, prefix caching tries to match the "prefix" of a prompt to those of all of the other prompts the model has processed before. If there is a match, the KV cache of the prefix part that is matched no longer requires recomputation. Instead, the cached data is retrieved—in simple cases, directly from GPU memory.

The KV cache can also be offloaded to other places, such as the CPU, local SSD, or distributed across GPUs; on completely different machines; or even on distributed external storage systems—but we will discuss more complex storage design in Chapter 7. For now, though, let's assume the KV cache is all saved in current GPU memory. In this simpler case, if prefix caching is not enabled, after each request is completed, all KV cache generated for that request is removed from GPU memory. If prefix caching is enabled, the requests' KV caches are kept in GPU memory as long as there is still space available. In most prefix-caching implementations, a least recently used (LRU) mechanism is used to evict the KV cache when not enough GPU memory is available.

RadixAttention

One of the prominent prefix-caching solutions is RadixAttention, which was introduced along with the SGLang serving framework. RadixAttention leverages a *radix tree,* which is a data structure that works like a *trie* (also known as a *prefix tree*): a form of string-indexed lookup data structure to keep track of the prefix strings for the KV caches. When the tree grows too large, it also uses an LRU mechanism to evict the KV cache, applied recursively on the tree's leaf nodes.

Figure 6-23 shows a very basic example of two requests that share the same head node, `You are a helpful assistant`, and then branch afterwards. When two requests share the same prefix, RadixAttention can find corresponding nodes in the tree structure, which is stored in CPU memory. With each node mapping to the KV cache, which is stored in GPU memory, it can reuse anything in the KV cache without incurring recomputation. As new requests come in over time, the tree grows by adding leaf nodes and gets trimmed by LRU when it becomes too large.

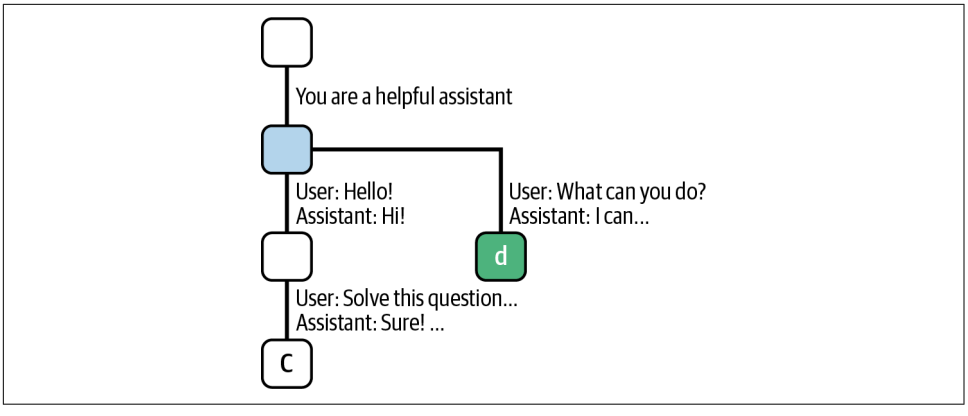

Figure 6-23. A radix tree example for two requests sharing the same prefix (source: Zheng et al. 2024)

Use Cases

Now let's examine in what cases a prefix cache works and in what scenario it becomes useful. Let's start with an example. Consider these three prompts:

```
Prompt 1: Hi, what is the weather like today?
Prompt 2: Hi, what is the weather like now?
Prompt 3: What is the weather like today?
```

Prompt 2 can reuse prompt 1's prefix KV cache from "Hi, what is the weather like…?" However, prompt 3, because it does not have the "Hi" as prefix, would need a complete fresh recalculation. You might wonder: does that mean it's not really very powerful? But in real-world LLM production, it is already proving very helpful indeed. Here are two major scenarios where prefix caching works at its best:

Scenario 1: Multiturn chat

One user interacts with an LLM, going back and forth multiple times. The prior chat history is not omitted when the user sends a new prompt. Instead, it is kept and concatenated with the new prompt the user sends. For example, the user first asks:

> *User*: What is the weather like today?
>
> *LLM*: The weather is 70F, but with a likelihood of rain.

The user sends a second question:

> *User*: Could you tell me the actual possibility?

The actual prompt sent to the LLM is not just that question alone. The LLM is usually given all the prior interactions as context, so that it can produce the best

response possible. The LLM knows that the user is not asking about an arbitrary possibility, but about the possibility of rain today.

Without prefix caching, the LLM would perform the prefill phrase for the first two lines again when the user asks the second question. *With* prefix caching, since the LLM has already processed the first two lines, they are kept in the KV cache in GPU memory instead of getting discarded. The LLM sees the match, knows that it has processed the first two lines before, and reuses the stored KV cache, without starting over from the very beginning.

This might look like a small reduction at first, but as the context of the multiturn chat grows longer and longer, without prefix caching, the user would experience longer and longer wait times when they ask additional questions. Most notably, the TTFT, which dominates the customer experience in LLM chatbots, would increase.

Scenario 2: Long context serving
: Another scenario where prefix caching provides critical performance gain is long context serving. LLMs' context size continues to grow, from 4,000 to 128,000 and even 1 million. This gives users a lot of opportunity to feed as much relevant information as possible into the input prompt, without always relying on extracting relevant information with RAG. But a very long context inside the prompt poses challenges in the prefill phase of LLM serving, which can make the TTFT intolerably long. Here comes prefix caching to the rescue: it saves the KV cache of all of the relevant information to avoid recomputation, as long as the user carefully reconstructs the prompt to ensure that its prefixes do not vary. This can result in cache hits every time.

Prefix caching is now enabled by default for almost all scenarios, even when the cache hit rate and TTFT improvement rate are not as good as the two mentioned here. This is because modern optimized serving engines are capable of achieving minimal or even zero overhead when turning prefix caching on. In other words, there is almost no drawback to using it. Even if the cache hit rate is only 5% in your use case, the blazing TTFT speed of that 5% of the customer experience is still worthwhile.

Best Practices

To best utilize prefix caching, your key target should be to improve the cache hit rate. The first and foremost way to do that is to carefully structure prompts so that the LLM cleanly separates the static parts (which target the prefix hit at the very front of the prompt) from the dynamic parts (which correspond to user input at the back as the suffix).

You can use the following template to construct prompts so that everything in the system and context results in cache hits:

```
<system>
You are a helpful assistant.
<context>
Document: {puts the relevant static context here each time}
<user>
{dynamic questions from user}
```

Even just changing the word *Document* to *Documents*—just one extra letter—can result in a cache miss. Thus, assembling the prompt programmatically and consistently is very important.

The same is true for RAG use cases. Even though these cannot have as high of a cache hit rate as the long-context scenario, you still want to achieve a high cache hit rate and make the hit as long as possible. For example, in RAG, your context would be shaped like this:

```
Document 1: <retrieved_text_chunk_1>
Document 2: <retrieved_text_chunk_2>
Document 3: <retrieved_text_chunk_5>
Document 4: <retrieved_text_chunk_7>
```

This version has clean formatting and sorted document chunks.

Instead of:

```
Document1: <retrieved_text_chunk_1>
Document 2:<retrieved_text_chunk_2>
Document 3: <retrieved_text_chunk_5>
Document 4: <retrieved_text_chunk_7>
```

The issue with this version is the formatting of the first two lines omitting extra spaces here and there can cause cache miss.

Or this:

```
Document 1: <retrieved_text_chunk_5>
Document 2: <retrieved_text_chunk_7>
Document 3: <retrieved_text_chunk_1>
Document 4: <retrieved_text_chunk_2>
```

The issue with this version is the ordering.

In summary, it is still important to maintain the same formatting and structure each time. Moreover, consistent ranking and proper deduplication can also help to achieve the longest prefix hits possible. For example, even if you can't get a prefix hit of the full context when retrieving the same first two chunks, you can still achieve a prefix cache hit all the way to "…Document 3:," which is still a win.

Scaling Prefix Cache

As your serving traffic increases with more requests and more customers, it is inevitable that you'll need to scale your setup.

First, let's consider a setup with only a single model instance. In Chapter 5, you saw in Figure 5-11 that KV cache can take up a significant percentage of GPU memory during LLM serving. With prefix caching, that's even more true, since it is advantageous to cache as many requests as possible to increase the prefix cache hit rate. Thus, reserving enough space in the GPU to cache common prefixes is important. (See Chapter 7 for more on other storage methods.)

Another unique challenge when using the prefix KV cache is that, as traffic increases, your setup will need to scale horizontally, to have more model instances working in parallel. To route and distribute requests efficiently to all the model instances, common load-balancing techniques are *round robin*, where requests get distributed across the servers in a sequential or rotational manner; *least connection*, where requests get distributed across the servers with the fewest active connections; or based on custom metrics, such as GPU utilization for ML workloads. With prefix caching, since the prefix KV cache is local, it becomes important to have a smart routing layer. Like consistent hashing, it creates an affinity between prefixes and mode instances, so that requests are routed to the model instance (or instances) that has the prefix KV cache already stored, rather than to a new instance to start the prefill process over. Figure 6-24 illustrates this setup.

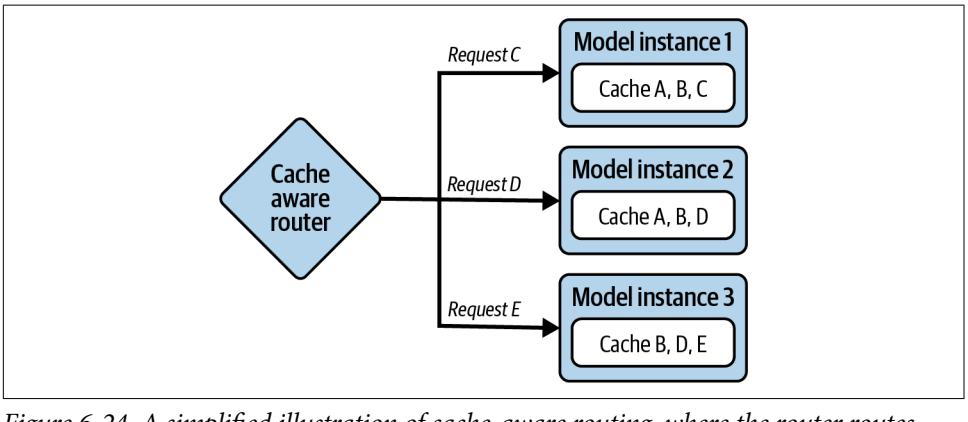

Figure 6-24. A simplified illustration of cache-aware routing, where the router routes requests to achieve high cache hit rates, based on the model instance's request prefixes and the prefix cache

With the routing layer, not all model instances need to cache all of the prefixes as well. This helps reduce the amount of GPU memory required and avoids constant evictions and recomputation. But what if the prefixes are very long or you need to cache a lot of different prefixes, and GPU memory still isn't enough? One solution is to offload the KV cache to CPU memory and/or SSD when needed (see Chapter 7).

Finally, the LLMs you are serving will usually be *multi-tenant*: in other words, different customers will share the same model endpoint and instance. It would be too

expensive to create one model instance for each customer, and batching different customers' requests together can greatly reduce the real-world batch size to improve GPU utilization and save cost. But, with prefix caching, customer A's prefixes might be the same as customer B's prefixes by accident. Customer A could then perform enumerations to infer the data processed by another customer, based on the latency it observes. This is a behavior we do not want.

One way to isolate prefixes to a single tenant is to inject a unique ID for each customer into the prompt, as NVIDIA discusses in its technical blog. In the following example, a user ID or session ID is added between the system prompt and the context section, so prompts from different customers can only share the same system prompt as a prefix when they have different IDs and no context gets shared across different customers:

```
<system>
You are a helpful assistant.
<id> {user id or session id}
<context>
Document: { puts the relevant static context here each time}
<user>
dynamic questions from user}
```

Summary

In this chapter, we delved into a spectrum of techniques to optimize the LLM serving performance in real-world deployment settings.

The chapter began by exploring online request-batching and scheduling techniques. By aggregating multiple incoming requests into batches, models can process them more efficiently, optimizing arithmetic intensity and overall throughput. Dynamic batching is a common strategy for general ML workloads. A more advanced technique, continuous batching, which addresses the LLM specifically, takes precedence over dynamic batching to continuously add new requests into the batch, reducing GPU idle time. Today, continuous batching is a default solution for LLM online serving. On top of that, to better balance TTFT and ITL and improve overall throughput, you can enable and tune chunked prefill to tackle very long contexts and interactive use cases.

Next, we looked at how to improve attention calculation. One way of doing that is to reduce KV cache size by sharing the same KV across multiple heads while maintaining accuracy in the evolution from MHA to MQA to GQA. Another is by performing smart compressions, such as MLA in DeepSeek. Then we explained how kernel optimization can improve attention computation performance, for example by doing kernel fusion to reduce round trips back to memory during calculation. We also looked at FlashAttention where, with techniques such as tiling, the algorithm smartly avoids using HBM. Instead, it keeps data in SRAM to avoid HBM bottlenecks

based on hardware IO specs and limitations. Finally, PagedAttention, which is relatively orthogonal to kernels for computation, focuses on optimizing GPU memory access by partitioning the KV cache into small blocks called *pages* to reduce memory fragmentation. This is now the default choice in KV cache management for LLM serving.

To further improve model serving performance, model compression trades some accuracy for significant size reduction and execution performance improvement. The most common technique used in industry is quantization—more specifically, post-training quantization (PTQ)—where after model training completes, model weights are quantized from high-bit to low-bit formats. We discussed quantization techniques and low-bit formats, along with their trade-offs. We also glanced at other model compression techniques, such as model distillation, where a large model can teach a small model, and model pruning, to reduce unnecessary and insignificant weights in the model.

Finally, we showed you how prefix caching, as an LLM-specific request-caching method, helps reuse partially seen requests, trading storage for less computation and very fast TTFT. Prefix caching excels at multiturn chat and long-context scenarios, which are becoming more and more essential in modern LLM applications.

Now you should be equipped with the essential knowledge to serve LLMs efficiently, especially smaller LLMs that run on a single GPU. In the next chapter, we will dive into advanced topics to help you run large LLMs in distributed fashion, and improve LLM serving as a system over focusing on a single model replica.

CHAPTER 7

Advanced LLM Optimization Techniques

After the last chapter, you are equipped with essential techniques to combat many of the challenges of LLM serving optimization, especially those that are not overly large and fit into one GPU. For larger LLMs with, for example, more than 100 billion parameters, one GPU is usually not enough to load the model to GPU memory and generate at a satisfactory latency. In this chapter, we explore advanced techniques to further enhance LLM serving performance, including:

- *Speculative decoding* to speed up the decode phase of LLM generation for faster inter-token latency (ITL)
- Multi-GPU and multi-node serving for large LLMs that do not fit or are not performant enough when running on a single GPU
- *Prefill-decode (PD) disaggregation* to decouple the prefill and decode phases and fine-tune their trade-offs independently
- *Advanced KV caching techniques* to achieve lightning-fast time to first token (TTFT) and a high cache hit rate

Speculative Decoding

What if a single technique could singlehandedly improve latency—especially ITL—by a factor of two or three? Meet *speculative decoding*, a novel approach that is particularly useful for long, reasoning-heavy generations.

In a large ML system with, say, a million data points for retrieval or recommendation, it's common to use a small but less accurate model to do the first round of filtering. Once you've significantly reduced the number of candidate data points to around

1,000, you then apply a larger, more accurate model on those remaining candidates to produce the final result.

Speculative decoding does something very similar, but at the token level. It leverages a small model, which we call the *draft model*, to help generate candidate tokens for the large "target model." This speeds up the decode phase of LLM generation. In other words, during token generation, we have our small draft model speculate on what the next tokens could be. Rather than generating tokens one by one, the target model now *verifies* the generated tokens. If the target model finds that the draft model is doing a decent job, it accepts the token and jumps ahead.

Detailed Steps

We'll start with one iteration of speculative decoding, where the draft model performs token speculation and the target model then performs parallel verification, as shown in Figure 7-1.

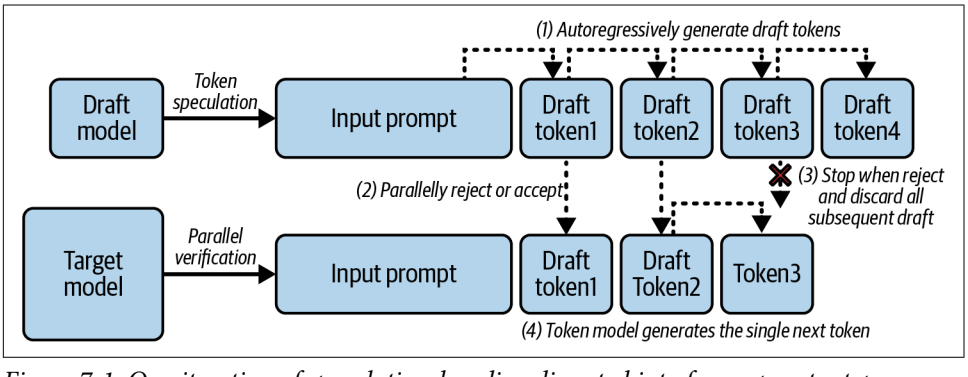

Figure 7-1. One iteration of speculative decoding dissected into four separate steps

First, the draft model quickly generates K tokens. K here represents the number of tokens the draft model speculates, which is an important parameter to tune for the optimal value. Immediately, the target model performs its forward pass, during which it verifies if the drafted tokens are good or not.

In token generation, the next token is generated with a probability value. For example, given the phrase `The soccer team of the United`, the draft model might speculate that the next token will be `States`, with a probability of 0.6, whereas the probability of `Kingdom` is 0.3 and that of `Nations` is 0.1. If our target model thinks `States` is 0.8 probable, which is higher than 0.6, it will accept the drafted token. If it thinks `States` is 0.4 probable, usually we calculate it probabilistically by dividing 0.4 by 0.6.

If the target model still accepts the draft token, it can move forward to verify the next token. If it decides to reject the draft token, we discard all the following draft tokens.

We do this because the generation is autoregressive, so none of the following tokens will be relevant. The candidate model will instead generate one token that it thinks appropriate. This is represented by Token3 in Figure 7-1.

Importantly, this process does not compromise accuracy. In other words, the final output of the speculative decoding process should be identical to what a nonspeculative autoregressive model would produce. This is achieved by the verification step in particular. When a token is rejected, the target model corrects the problem by sampling a new token from its own modified distribution, which it adjusts to exclude the rejected token. The process restarts from this new, confirmed point.[1]

Tuning and Usage

This section provides some important tips for applying speculative decoding in real-world use cases.

Choose an existing small model

To maximize the performance of speculative decoding, the key is to use a small model that maximizes both speed and acceptance rate. Using a model from the same family means the same tokenizer and pretraining can really help in achieving a high acceptance rate. It's also a good idea to be aggressive in quantizing your draft model, since you always have the target model as the fallback.

If you can perform some training, distilling a small draft model from your target model is usually a better choice than picking an existing small model. This alignment reduces the mismatch between the drafter's predictions and the target's verification step, which in turn increases the acceptance rate of drafted tokens and improves overall efficiency. In practice, this makes speculative decoding more robust, since it specializes the small model for the target's style and domain, rather than relying on general pretraining alone.

Use self-drafting

To push performance further, one idea, called *self-drafting*, moves away from relying on external draft models and instead lets the model draft its own future tokens. For example, self-drafting techniques like Medusa and EAGLE modify the target model with lightweight prediction heads or auxiliary modules to produce multistep lookahead inside the same model. Speculating without extra draft models saves GPU memory, simplifies deployment, and achieves good alignment with the target model.

1 To dive deeper into the mathematics behind speculative decoding and the modified distribution, refer to the original research papers: "Fast Inference from Transformers via Speculative Decoding" (Leviathan et al., 2022) and "Accelerating Large Language Model Decoding with Speculative Sampling" (Chen et al., 2023).

Figure 7-2 shows an example of self-drafting with Medusa. On the left, a forward pass with the original model only generates candidates for the first token, shown as `It, I, As` on the top right. What's new is that each Medusa head in the middle performs extra generations of the second, third, and fourth subsequent token candidates in parallel in the same forward pass. These many candidate tokens across each token position, from first to fourth, can be assembled into many candidate sequences, from which the longest accepted candidate is chosen. In the case shown in Figure 7-2, one iteration of the model's forward pass generates three tokens (`It is difficult`). Without speculative decoding, this pass generates only one token (`It`).

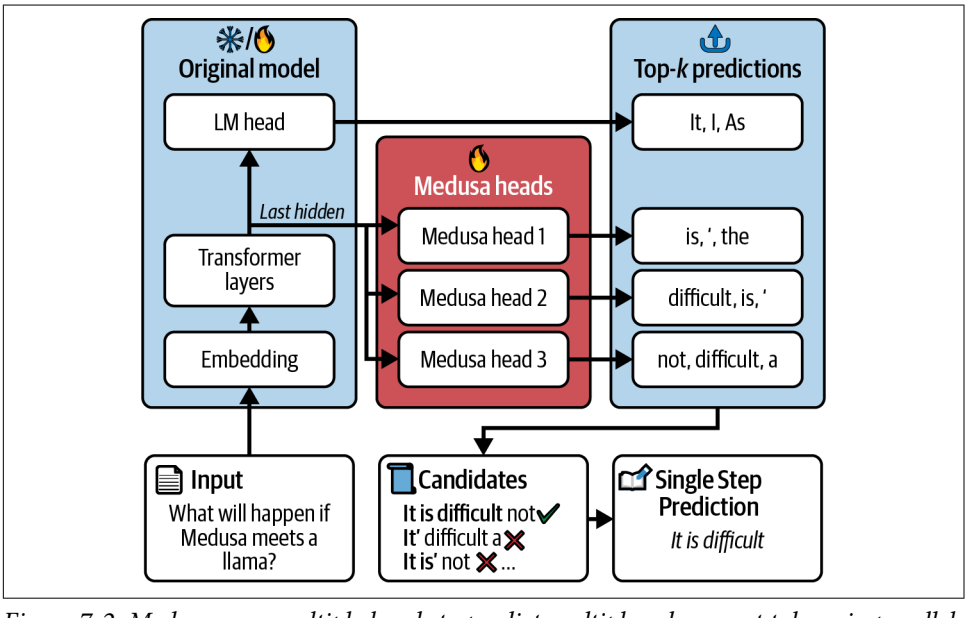

Figure 7-2. Medusa uses multiple heads to predict multiple subsequent tokens in parallel (adapted from Cai et al., 2024)

Extrapolation Algorithm for Greater Language-Model Efficiency (EAGLE), however, does not draft tokens directly like Medusa does. It predicts hidden states instead. To be more exact, it uses a small auxiliary module that is trained to predict the target model's future internal hidden states instead of draft tokens. The target model then generates tokens from these predicted internal hidden states. This results in more stable and accurate predictions. Later on, EAGLE-2 introduced a dynamic draft tree to adapt the length of its speculation based on how predictable the current text is, and EAGLE-3, shown in Figure 7-3, further enhanced accuracy by fusing features from multiple LLM layers as input. As we write this at the end of 2025, EAGLE is one of the highest-performing self-drafting techniques. To perform well, however, it requires extra training and tuning.

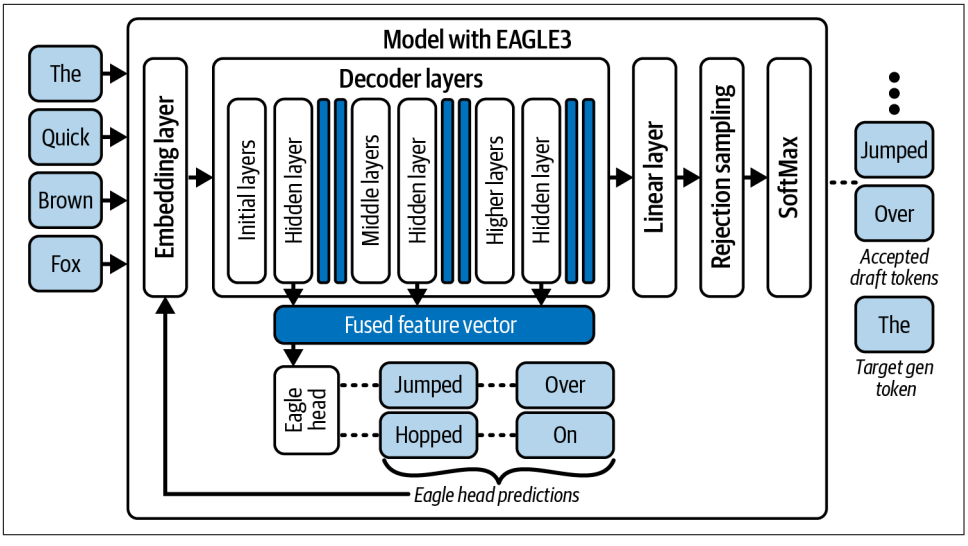

Figure 7-3. EAGLE-3 utilizes fused feature vectors from multiple layers for draft token generation (adapted from Li et al., 2025)

N-gram

Instead of having a small model or module speculate about tokens, there's a third option: a simple method called an n-gram. An *n-gram* basically chooses a sequence of *n* adjacent tokens from the earlier part of the request, saves them as a n-gram table, as shown in Table 7-1, and performs matching to propose the next token(s). For example:

```
An hour ago, a quick brown fox ran away and now, that quick brown [next
token]
```

Now you need to predict the next token. We capture this example sequence in a cached n-gram table to propose that the next token will be `fox`, based on the second row of the trigram table (Table 7-1). There's no need for extra complex predictions. The original model takes `fox` and verifies it.

Table 7-1. Example rows of an n-gram table where n = 3, captured from the example sequence

Trigram table (n = 3)	Next token	Count
a quick	brown	1
quick brown	fox	1

The n-gram method without an auxiliary model has very low computational overhead and can be very powerful, particularly in two scenarios. The first is when you are generating structured outputs such as JSON or SQL files, where the generation follows highly deterministic and repetitive patterns. The second is when asking the LLM to improve a piece of writing or fill a preset template—situations where the LLM can quickly reuse many n-gram sequences from the original prompt, as shown in the following example.

Here is the original input prompt:

```
Write a brief confirmation email to a customer whose name is Iris for a
Tuesday 2pm onboarding call. Keep it professional and friendly.
Use the following email template:

Subject: <one line>
Greeting: Hi <name>,
Body: Confirm the onboarding call date and time; mention sending a calendar
invite with a video link; offer to reschedule if needed.
Closing: Best regards
```

Here is the model's generated response:

```
Subject: Onboarding Call Confirmation - Tuesday 2:00 PM

Hi Iris,

This is a quick note to confirm our onboarding call on Tuesday at 2:00 PM
Pacific. I'll send a calendar invite with the video link shortly. If anything
changes, just reply here and we'll find a time that works.

Best regards,
Andy
```

In this case, we are providing a lot of predefined structure in the prompt, which will help to achieve a higher acceptance rate. That, in turn, allows us to push for higher *K* and achieves lower latency.

Because n-grams incur minimal overhead, even if the acceptance rate is low, they are still a very viable option. We recommend making this the first speculative method you try before the other draft models.

Tune the number of draft tokens

The other crucial knob for speculative decoding is *K*, the maximum number of draft tokens. When *K* is large, it gives a very high ceiling for speedups. However, if the acceptance rate is low, it's less effective, since many draft tokens are discarded and wasted. Conversely, if *K* is small, there are fewer tokens to verify and reject, so you get more predictable performance, but could be underutilizing speculative decoding and not getting its full benefit. In general, *K* values are something to tune on. They depend highly on the use case, the target models, the drafting method's overhead, and

the performance SLOs you want to achieve. For example, because the n-gram method has very low overhead, it can tolerate a lower acceptance rate and still provide a net gain in performance.

A smaller K, such as 2 or 4, can form a decent and stable baseline from which you can increment. In many cases, the best K is somewhere between 4 to 8. For very predictable generations, such as structured outputs or AI-agent function calling, K can be quite a lot larger, even pushing toward 16 or 32.

In many LLM serving frameworks, you can observe the acceptance rate for each position. For example, if $K = 6$, you might get [0.8, 0.7, 0.6, 0.5, 0.10, 0.02]. This means the first token has an 80% acceptance rate, while the fifth and sixth tokens have rates lower than 10%. In this case, you might want to consider lowering your K, since the last two speculations are not yielding much gain.

Limitations

Speculative decoding *only* helps with the decode stage. Furthermore, the way it leverages a small draft model could result in extra computation when the acceptance rate is low. And if your scenario has a long input context in the prefill stage, meaning that the model is already bottlenecked on compute FLOPS, speculative decoding might not work.

Likewise, processing large batches can shift the bottleneck from memory-bound to compute-bound, as we discussed extensively in Chapter 6. Even if your scenario does not have a very long context and generates a lot of tokens, efficient batching can still move the bottleneck from memory-bound to compute-bound. If so, speculative decoding can still help with latency, but the extra computation will hurt overall throughput.

Productizing speculative decoding can be quite tricky. Not only is performance not guaranteed, but running two models on a shared GPU efficiently and with high resource utilization is not a simple task. This is why speculative decoding has recently begun to focus more on self-drafting and n-gram techniques than on utilizing a small model. Also, a static K might not be able to capture the changing demand. There has been a lot of research in this area aiming to make it more adaptive and efficient.

In short, the best scenario to utilize speculative decoding is when your workload is latency-sensitive and you are OK with trading some throughput and/or TTFT for better ITL and overall latency. When the prefill:decode token ratio is low and real-world batches are small, causing a GPU-memory-bound bottleneck, you can take advantage of speculative decoding's blazing speed to meaningfully improve the customer experience.

Hands-on Speculative Decoding

Let's try some speculative decoding examples in the Google Colab notebook using both n-gram and EAGLE-3.

Enable speculative decoding

The following code sets up an example command vLLM server with four different variations: vanilla vLLM as baseline, n-gram, an improved n-gram setup aimed to improve acceptance rate and lower overhead, and an EAGLE-3 setup:

```
# vanilla vllm
nohup vllm serve Qwen/Qwen3-32B \
 --disable-log-requests \
 --max-model-len 2048 \
 --gpu-memory-utilization 0.95 \
 > vllm.log 2>&1 &


# n-gram
nohup vllm serve Qwen/Qwen3-32B \
 --speculative-config '{
   "method": "ngram",
   "num_speculative_tokens": 6,
   "prompt_lookup_min": 4,
   "prompt_lookup_max": 6
}' \
 --disable-log-requests \
 --max-model-len 2048 \
 --gpu-memory-utilization 0.95 \
 > vllm.log 2>&1 &


# improved n-gram
nohup vllm serve Qwen/Qwen3-32B \
 --speculative-config '{
   "method": "ngram",
   "num_speculative_tokens": 4,
   "prompt_lookup_min": 2,
   "prompt_lookup_max": 128
}' \
 --disable-log-requests \
 --max-model-len 2048 \
 --gpu-memory-utilization 0.95 \
 > vllm.log 2>&1 &

# eagle3
nohup vllm serve Qwen/Qwen3-32B \
 --speculative-config '{
   "method": "eagle3",
   "model": "RedHatAI/Qwen3-32B-speculator.eagle3",
   "num_speculative_tokens": 3
```

```
}' \
--disable-log-requests \
--max-model-len 2048 \
--gpu-memory-utilization 0.95 \
> vllm.log 2>&1 &
```

Run benchmarks

Inside the notebook, we run two small benchmarks: one with a concurrency of 1 to simulate performance under low RPS, and the other with a concurrency of 16 to simulate a high-load, high-RPS scenario over the four setups.

Performance analysis

Start with the concurrency = 1 results on the left side of Figure 7-4, where you can observe that n-gram slightly improves the token throughput. The improved n-gram provides a gain of about 16% over vanilla vLLM. In comparison, EAGLE-3 achieves close to double vanilla vLLM's throughput (56.5 tokens/s versus 28.9 tokens/s).

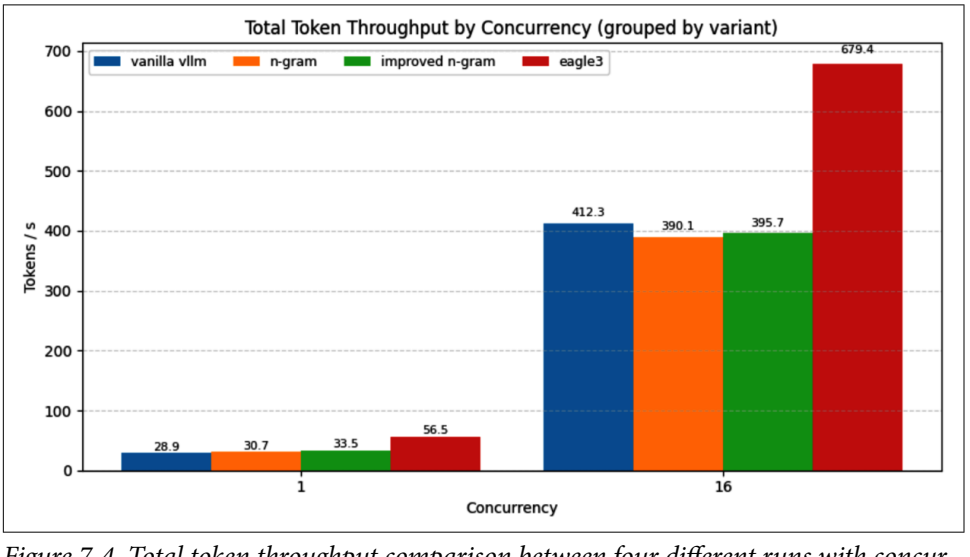

Figure 7-4. Total token throughput comparison between four different runs with concurrency of 1 and 16

Moving to concurrency = 16 on the right of Figure 7-4, now both n-gram runs perform worse than vanilla vLLM. This is due to the overhead incurred with proposing and verifying new tokens. This causes extra CPU and GPU compute load that becomes even more substantial at high concurrency settings. Similarly, though EAGLE-3 still significantly outpaces vanilla vLLM, the gain is smaller.

To help you understand this further, Figure 7-5 shows the TTFT and ITL across four runs under a high RPS scenario (concurrency = 16). As expected, both n-gram variants and EAGLE-3 perform better than vanilla vLLM. However, EAGLE-3's TTFT is significantly longer than that of vanilla vLLM, due to the overhead of the extra speculative heads and modules. Thus, the trade-off is between prioritizing TTFT or ITL, based on your SLA or SLO.

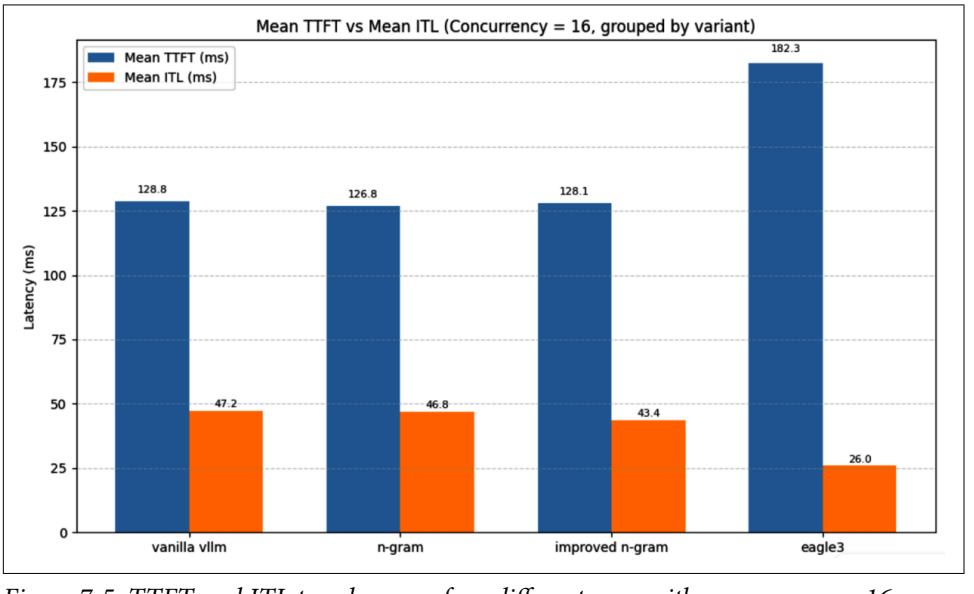

Figure 7-5. TTFT and ITL trends across four different runs with concurrency = 16

In real-world settings, the setup will be highly dependent on the dataset, and you will need to perform multiple runs with understanding of the acceptance rate across K to select the decode method and the best value of K.

Multi-GPU and Multi-Node Inferencing

Modern LLMs often exceed the memory footprint of a single GPU and serving them with low latency and at scale requires far beyond what one GPU can sustain. To overcome these limitations, inference systems turn to distributed strategies that harness multiple GPUs across one or more nodes. In this section, we shift our focus to distributed serving and discuss four parallelism techniques:

Data parallelism (DP)
: For scaling throughput: works by replicating model instances across GPUs and nodes

Tensor parallelism (TP)
: For fitting large models: works by splitting layers and large matrix operations to achieve lower latency

Pipeline parallelism (PP)
: For fitting large models: works by partitioning layers into stages across different GPUs and nodes

Expert parallelism (EP)
: For mixture-of-experts (MoE) models: works by distributing experts across GPUs

The following sections look more closely at each technique.

Data Parallelism

In software engineering, when the request count becomes too high, we horizontally scale the number of instances or replicas first. In the ML world, the preferred technique is very similar: we replicate the number of model instances so that each model instance can process a percentage of the total traffic. Both techniques are called *data parallelism*. DP is essential to scaling to handle large numbers of requests and ensure high availability. It is applied everywhere when handling production traffic.

Figure 7-6 shows a simple example of DP, where a load balancer/router accepts nine requests and spreads the load to three different model instances evenly. Each model instance gets three requests, so none of them is overloaded.

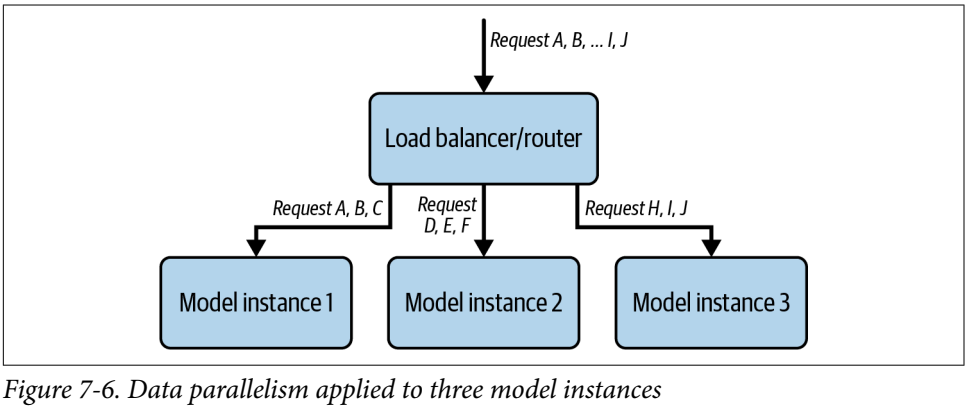

Figure 7-6. Data parallelism applied to three model instances

If one of the model instances goes down, the router can stop routing requests to it and send them to the other two, as shown in Figure 7-7. (This is a simple example; in practice, the routing strategy can be much more complex than that.)

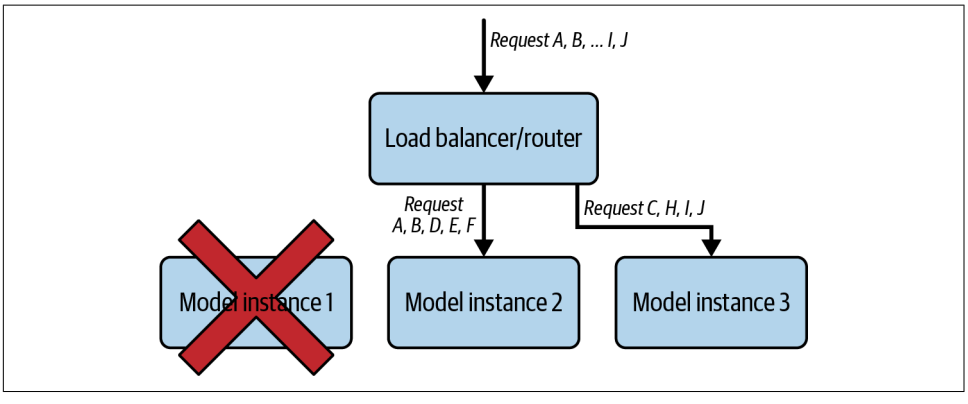

Figure 7-7. In data parallelism, when one model instance is down, the router routes requests to the two other running model instances

Common routing strategies used in DP include:

Round robin
: The simplest method. Requests are sent to each model instance in a fixed, sequential order. While easy to implement, it can lead to uneven loads because some requests may take much longer to process than others, leaving some instances idle while others are overloaded.

Least connections
: This strategy routes new requests to the instance with the fewest active connections. It's an improvement over round robin since it accounts for the current workload, but it doesn't consider the complexity or length of the requests already being processed.

Latency-based routing
: More sophisticated load balancers continuously monitor the real-time latency of each instance. Requests are routed to the fastest-responding instance. This method helps to reduce tail latency by dynamically adapting to performance fluctuations.

Cache-aware routing
: In Chapter 6, we introduced prefix caching, a powerful technique to reduce TTFT by reusing existing KV caches across requests when prefixes are the same. As you scale to multiple model instances, the key to achieving a high prefix-caching hit rate is to route requests with the same prefixes to the model instances that already hold those caches (see Figure 6-23 for a refresher).

Because LLM-serving workloads vary greatly from one request to another, and the underlying hardware for serving LLMs is extremely expensive, the value of an efficient routing mechanism is significantly higher than in traditional web applications. LLM routing has evolved beyond simple rule-based matching into a complex system that leverages real-time signals, such as:

- KV cache locality
- KV cache hit percentage (the ratio of number of tokens of KV cache hit to total prefix length)
- Model instance's KV cache space usage percentage
- Raw input length
- Number of requests queued that are waiting for processing for the model instance
- Number of pending requests that are in the middle of processing for the model instance
- Number of tokens already generated for the requests in progress
- Request SLA/SLO latency budget (the maximum acceptable response time to meet service guarantees)

To learn more about routing, see KV Router by NVIDIA Dynamo and the llm-d router documentation. These routers also support additional optimizations, such as prefill/decode worker routing and KV offloading, which we will cover in later sections of this chapter.

Tensor Parallelism and Pipeline Parallelism

When the data in a database grows too large for a single machine, we perform *sharding* to split the database into multiple shards, or pieces, across multiple machines. Similarly, when you need to perform inference with very large LLMs, a single GPU often won't be enough to load and serve the model fast enough to meet your SLA. This is where multi-GPU and multi-node inferencing come in. These techniques involve distributing a single large model across multiple GPUs, which can be in a single machine (multi-GPU) or sharded across several interconnected machines (multi-node).

Tensor parallelism (TP) and *pipeline parallelism* (PP) are two common methods for sharding a model across GPUs. TP shards the model along its width, while PP shards it along its depth, as shown in Figure 7-8. More specifically, TP splits the individual layers of a model *horizontally* across multiple GPUs. Each GPU holds a slice of a layer's weight tensor and performs a partial computation. The results are then combined to produce the full layer output. Pipeline parallelism, in contrast, splits the

model's layers *vertically* across multiple GPUs rather than splitting any single layer into multiple pieces. Each GPU is responsible for a contiguous block of layers. Data flows sequentially through these GPUs, like an assembly line: one GPU computes the first few layers, then passes the intermediate results to the next GPU, and so on, until the final output is produced.

Figure 7-8 compares the two approaches. With TP set to 2, each layer of the model is split in half parts, with each half running on one of the two GPUs. With PP, instead of splitting each layer, different layers go to different GPUs. In the diagram, with four layers and PP set to 2, layers 1 and 2 are located in one GPU and layers 3 and 4 in the other.

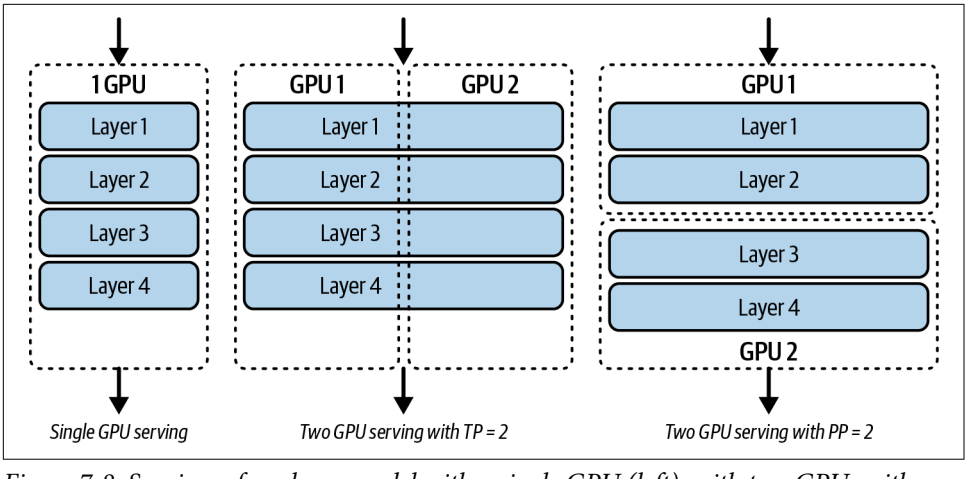

Figure 7-8. Serving a four-layer model with a single GPU (left), with two GPUs with TP = 2 (center), and with two GPUs with PP = 2 (right)

In Figure 7-9, diving deeper, tensor parallelism needs inter-GPU (between GPU 1 and GPU 2) communication for every single layer. Pipeline parallelism only needs inter-GPU communication once it finishes layer 2 and moves on to layer 3.

Pipeline parallelism minimizes inter-GPU communication in processing, which is a good thing. Does this mean it's better than tensor parallelism? It's not quite that simple. Like a factory assembly line, PP can form bottlenecks that delay progress. Inside the LLM, workloads often vary across layers. When one stage becomes slow, other subsequent stages may be waiting, and devices become underutilized. This is called a *pipeline bubble*. For example, if layer 2 is very slow, layers 3 and 4 can be waiting for work. So GPU 1 could have very high GPU utilization while GPU 2 is idling most of the time.

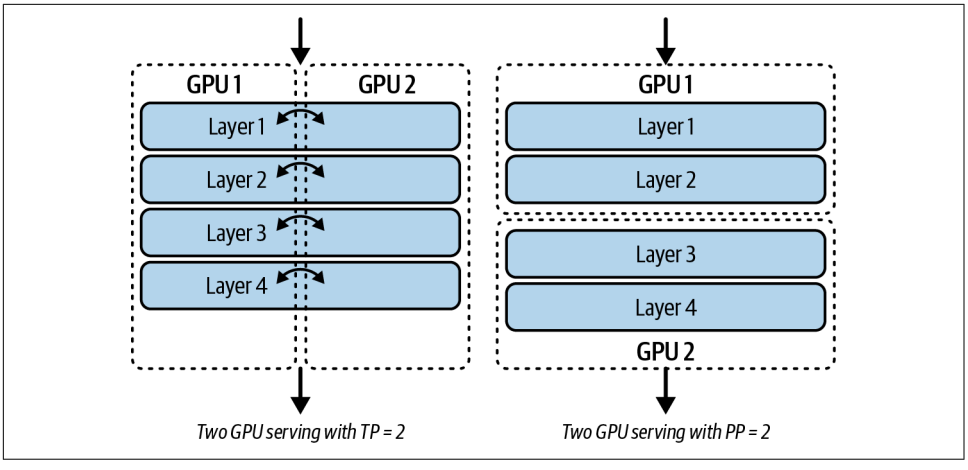

Figure 7-9. Cross-GPU communication in tensor parallelism (left) and pipeline parallelism (right)

Now, let's go through the decision flowchart in Figure 7-10 to better understand when to use these techniques and balance the trade-offs between TP and PP.

Starting from the top of the figure, the very first decision is whether you really need multi-GPU serving. You learned in Chapter 5 that GPU memory bandwidth is much faster than GPU interconnect. For example, H100 can transfer about 3 TB/s of data from GPU memory to the compute core; even with NVLink, the bandwidth is only ~900 GB/s across GPUs, which is around three times slower. This wide performance gap in data transfer speed results in several considerations. First, whether you have high-speed interconnects like NVLink or not, if your workload *can* stay within one GPU, it is still better to keep it in one GPU and avoid communication overhead. Second, adding GPUs does not scale linearly: for example, adding one GPU to the existing one doubles your GPU memory space and GPU FLOPS, but does *not* double performance because of the interconnection bottleneck. Third, in many cases, getting a higher-end GPU with a bigger memory so that you can host the model in a single GPU is better than getting multiple lower-end GPUs with parallelism.

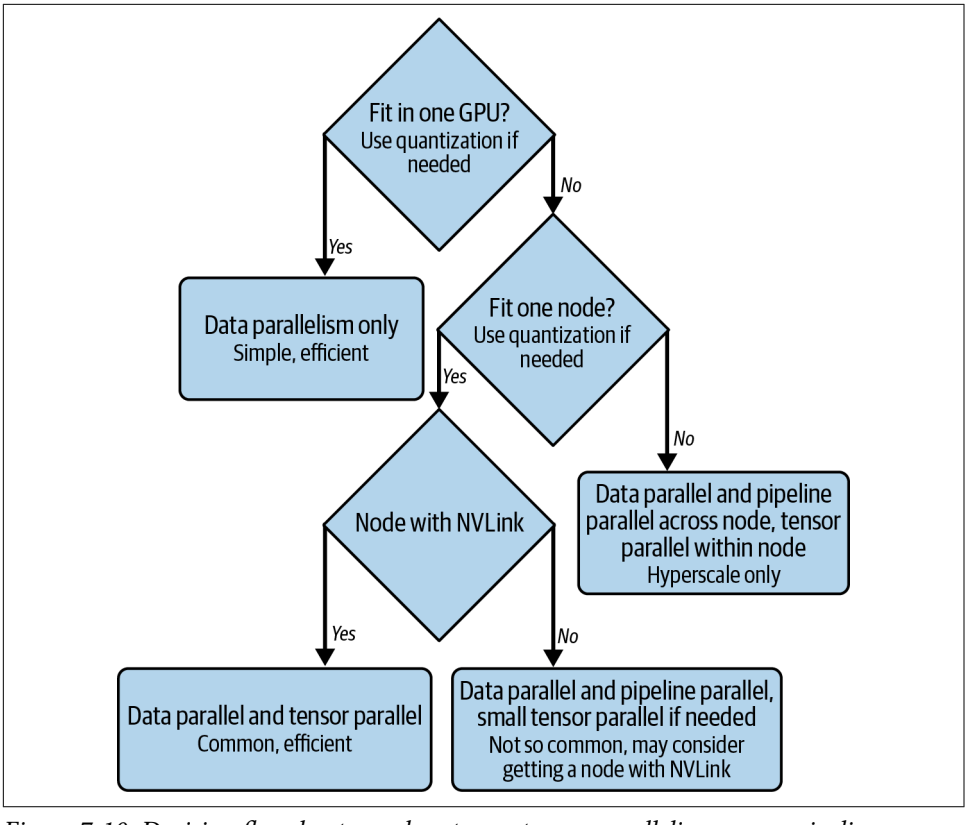

Figure 7-10. Decision flowchart on when to use tensor parallelism versus pipeline parallelism

Another key consideration is that if you are serving a model that won't fit in your GPU memory, don't forget that you can use quantization, as we discussed in Chapter 6. For example, FP8 quantization cuts model size in half compared to FP16 or BF16, and W4A16 quantization can bring it down to a quarter of the original size, if accuracy allows.

Now suppose that, even with quantization, your model still does not fit into a single GPU's memory or that a single GPU still isn't powerful enough to meet latency SLA. The next step would be to consider getting a node with multiple interconnected GPUs. For example, one AWS `P5.48xlarge` node has eight H100 GPUs, interconnected with NVLink, and one AWS `P5.4xlarge` node has one H100 GPU. One might think that getting eight `P5.4xlarge` nodes would be equivalent to a single `P5.48xlarge`, but this is not the case. This is because, within the `P5.48xlarge` node, the eight H100 GPUs can talk to each other via NVLink. However, for the setup with eight `P5.4xlarge nodes`, the eight H100 GPUs talk to each other *across*

nodes, through technologies such as InfiniBand, which is multiple times slower than NVLink.

With one node containing multiple GPUs, you can start applying TP or PP so that you can run your model across multiple GPUs. You've seen that PP minimizes the GPU interconnect load, but can cause pipeline bubbles. TP does not have this issue, but requires a lot of inter-GPU data transfer. Because of that, TP is the ideal choice if your GPUs within the node are connected with NVLink (for example, you're running a big LLM on eight GPUs on `P5.48xlarge`—a pretty common setup). However, If you don't have NVLink, current PCIe (which is designed for device-to-host traffic, not for high-speed GPU communications) has transfer speeds that are usually too slow for TP to work well. In that case, consider switching to a different instance or trying PP. Use zero TP or a very small TP to minimize inter-GPU communication here. In general, this scenario is less common and not really ideal.

On the very right of Figure 7-10, you can see the *hyperscale setup*. Here, the model not only doesn't fit into a single GPU, it doesn't even fit into a single node with eight high-end GPUs, either. In these cases, you can pull in multi-node serving. TP is usually used within nodes with NVLink to ensure fast inter-GPU, intra-node performance, and at the same time PP is used across nodes since inter-node communication is slow. Again, this is where pipeline parallelism shines: it minimizes the required data transfer across nodes. More advanced techniques such as prefill-decode disaggregated serving can also be very helpful here, which we will dive deep into in the next major section in this chapter. Figure 7-11 shows an example of sharding a model both on width (TP = 8) and height (PP = 2).

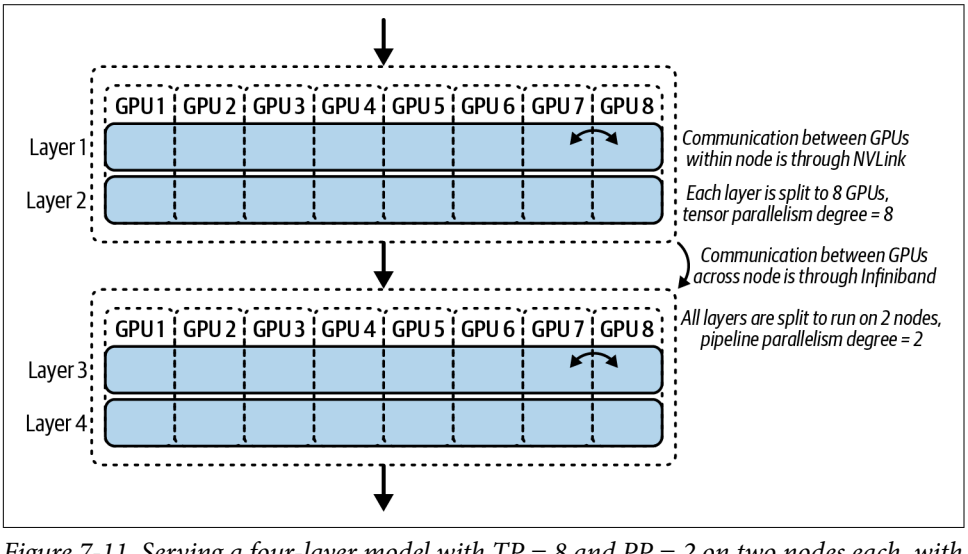

Figure 7-11. Serving a four-layer model with TP = 8 and PP = 2 on two nodes each, with eight GPUs

This snippet is a simple example of enabling TP and PP within a single eight-GPU node:

```
hf_model_id = "Qwen/Qwen2.5-7B-Instruct"
!vllm serve \
     --tensor-parallel-size 4 \
     --pipeline-parallel-size 2
```

Running vLLM across nodes is a bit more complicated and less used because of the network bottleneck. If your workload requires multi-node inference, you can use Ray with vLLM, a distributed computing framework that helps manage the distributed execution of tasks across multiple nodes.

Expert Parallelism

Modern LLMs often go beyond splitting a single dense model, using MoE layers designed to dynamically route tokens to different experts. Models such as Mixtral 8x7B, DeepSeek-V3 and GPT-OSS leverage MoE to reduce serving latency and cut serving costs while still delivering the performance characteristics of much larger, denser models. Figure 7-12 is a very simple illustration of how MoE works. To learn more on MoE, we recommend Maarten Grootendorst's newsletter.

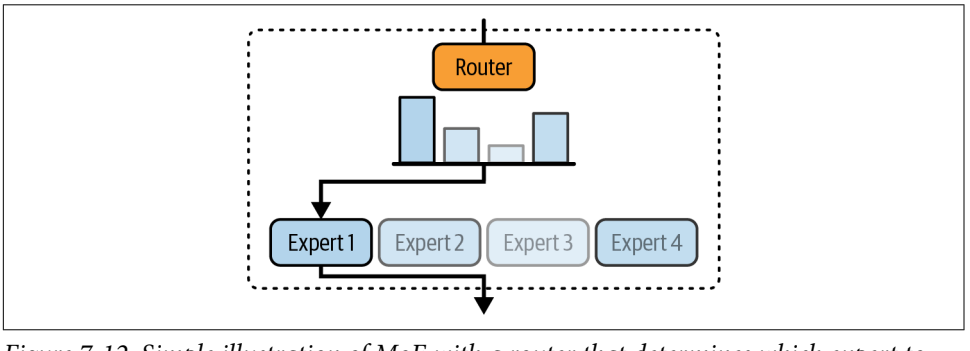

Figure 7-12. Simple illustration of MoE with a router that determines which expert to use (source: Grootendorst, 2024)

While MoE architecture reduces serving-time computation by activating only a few experts per token, it introduces a new challenge: the total parameters can become too large to fit on a single GPU. Ideally, experts can be distributed efficiently across hardware. This is where expert parallelism (EP) comes in: it partitions the experts across multiple GPUs so that routing and execution can scale seamlessly, as shown in Figure 7-13. In practice, EP complements TP and PP to achieve the best performance together by partitioning experts evenly across GPUs. So, for each incoming token, requests are dispatched only to the corresponding GPUs where the selected experts are loaded, avoiding unnecessary computation on inactive experts.

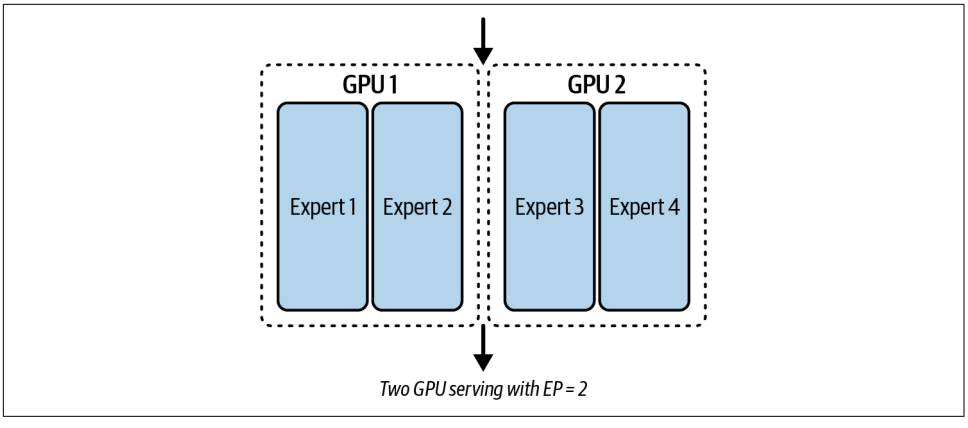

Figure 7-13. Simple illustration of EP = 2 for a four-expert model (other, non-MoE layers are not shown)

Prefill-Decode Disaggregation

Tensor parallelism and pipeline parallelism mainly address how to fit and accelerate a single large model with multiple GPUs. However, they don't solve inefficiencies that come from the two-phase nature of inference itself. Recall that the prefill stage processes the entire input sequence in parallel and is compute-intensive, whereas the decode stage generates tokens sequentially and is dominated by memory bandwidth. In all of the setups we've looked at so far, these two stages are scheduled on the same GPU or GPUs. This colocation usually means that the GPU needs to process one phase, either prefill or decode, while the other waits. In Chapter 6, you learned that the chunked prefill technique can mitigate this issue by splitting long input prompts into smaller chunks, thus unblocking the decode phase from waiting for long prefill workloads to complete. However, processing prefill and decode at the same time on the same set of GPUs still presents a fundamental challenge: the two stages have vastly different resource-utilization profiles. This can cause high interference and makes it difficult to optimize for both simultaneously.

Thus, separating the prefill and decode stages into different GPUs has become a popular method for large models and large-scale serving systems. By physically separating the compute-heavy prefill phase from the memory-and-bandwidth-heavy decode phase, *prefill–decode (PD) disaggregation* provides several important benefits:

Optimize TTFT and ITL independently
: On the user-experience side, PD disaggregation allows you to independently optimize TTFT or ITL by scaling the resources for prefill and decode asymmetrically, as shown in Figure 7-14. Depending on the use case, for example, the input/output token ratio can vary a lot. For an input-heavy, interactive workload, TTFT is key. PD disaggregation makes it easier to add more resources dedicated to

the prefill stage to complete long input prompts much faster, reducing wait time for TTFT. You can do the same for decode later on, if ITL needs improvement. Moreover, because there is no interference due to having prefill and decode on the same compute resource, ITL can be stabler and more predictable.

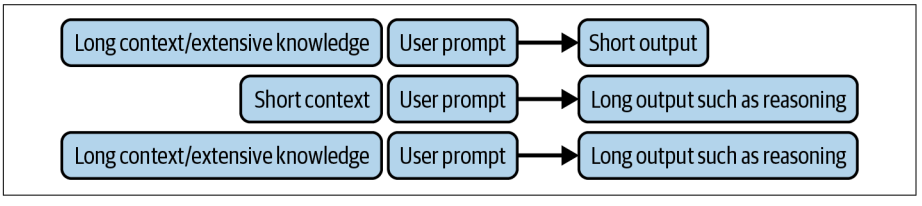

Figure 7-14. Different use cases can have very different input (prefill) and output (decode) ratios, leading to different TTFT/ITL trade-offs and requirements

Optimizing the prefill and decode workloads independently

On the GPU-utilization side, PD disaggregation lets you completely separate the prefill and decode workloads. The high-compute prefill stage does not sit idle waiting for the memory-bound decode stage to finish, and vice versa. This separation also lets prefill and decode have different batch sizes.

For prefill, depending on the input size, you can adjust the batch size to produce large enough matrix multiplications to saturate GPU tensor core utilization. For decode, you can tune the batch size to be large enough to ensure that the workload is no longer memory-bandwidth-bound while balancing the ITL latency requirements. Similarly, you can apply a heterogeneous parallel strategy: for example, you might apply TP and PP differently to prefill or decode to further improve or fine-tune the performance of one stage without affecting the other.

Opening up more hardware options

On the hardware selection side, PD disaggregation also means more flexibility. Prefill can use more compute-optimized GPUs for more FLOPS over cost, while decode can be scheduled on memory-optimized GPUs, with high memory bandwidth over cost. For example, the H200 GPU and H100 GPU have the same FLOPS specifications, but the more expensive H200 has more GPU memory (141 GB over 80 GB) and GPU memory bandwidth (4.8 TB/s over 3.35 TB/s). In this case, the H100 could be more cost-effective for the prefill stage, while the H200 would be better for decode. If you don't need that much performance for decode, you could use the L40S instead, which has decent GPU memory for a much lower cost. In this example setup, using the H100 for prefill and the L40S for decode, you can achieve an optimal TTFT and satisfactory ITL while saving a significant amount of money.

Enabling independent and asymmetric scaling

PD disaggregation offers fine-grained control of the prefill and decode stages, which means their scaling can be more elastic to deal with fluctuating demand. Prefill, as the entry point of the requests with varying input length, can be more burst-driven and can be paired with a more aggressive scaling strategy. Decode generates tokens over a more prolonged period and is more predictable and stable.

Overall Architecture

Now let's look at how it's done. Figure 7-15 shows the original DistServe runtime system architecture. All incoming requests arrive at the controller for routing. Requests are first sent to prefill instances for the prefill stage. The prefill instances produce the KV cache, which is immediately transferred to the decode instances, which then generate tokens to complete the requests.

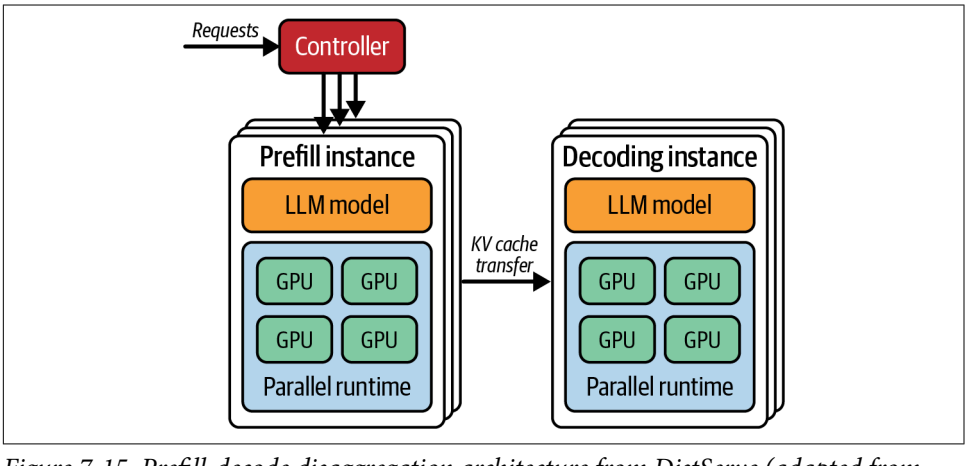

Figure 7-15. Prefill-decode disaggregation architecture from DistServe (adapted from Zhong et al., 2024)

Figure 7-16 shows that in traditional aggregated serving, GPU instances are tasked to do both prefill and decode, consistently balancing the compute-bound and memory-bound workloads. In disaggregated serving, however, GPU instances are split into two instance groups, with dedicated prefill instances to maximize computation and decode instances to tackle memory-bandwidth issues.

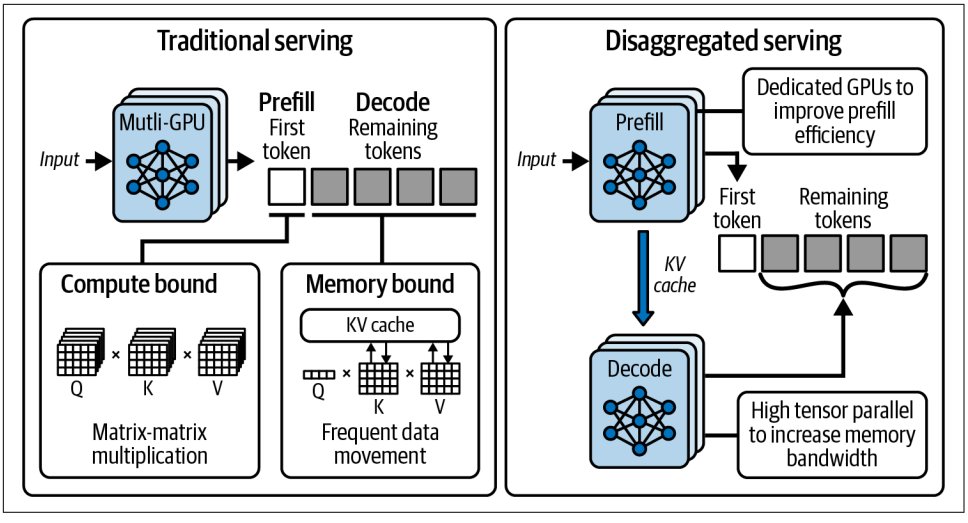

Figure 7-16. Side-by-side comparison of traditional aggregated serving and PD disaggregated serving (adapted from Elmeleegy et al., 2025)

KV Cache Transfer

One important step we have not yet covered, but which you might have noticed in both Figures 7-15 and 7-16, is the KV cache transfer. Efficiently transferring the KV cache from the prefill instances to the decode instances is critical to the success of disaggregated serving. The performance gain from PD disaggregation needs to outweigh the overhead of the KV cache transfer.

As we discussed in Chapter 5, GPU memory is scarce and its most significant usage, apart from model weight, is for KV cache, which is essential for fast decoding when you trade memory space for less computation. Despite technological advances and compression and quantization techniques to reduce its size (see Chapter 6), the KV cache can be very large. For example, a common 8-billion-parameter model with 1,024 input tokens can have a KV cache size of around 0.1 to 0.15 GB. This might not sound like a lot, but consider that a GPU is not processing one request at a time. As continuous batching pushes the throughput higher, one GPU can process tens of requests or more per second. Moreover, in real-world applications, input length can easily be *multiples* of 1,024. The KV cache size to be transferred scales mostly linearly with input length. Thus, 10x means KV cache size per request becomes 1 to 1.5 GB. Now multiply that by a throughput of, say, 16 requests per second, and you need to be able to transfer close to 25 GB of KV cache per second.

With NVLink, intra-node communication bandwidth performance can be as high as 900 GB/s or more, whereas inter-node, even with InfiniBand, is around 50 GB/s to 100 GB/s. (See Table 5-3 for a comparison of intra-node and inter-node communication bandwidth.) Without RDMA technology like InfiniBand, inter-node connection needs to go through PCIe, which is usually around 10 GB/s. Thus, how to allocate the GPUs for prefill and decode based on your hardware setup becomes important.

This example's 25 GB KV cache transfer is well within the capabilities of intra-node NVLink (900 GB/s) and works OK for inter-node InfiniBand (50–100 GB/s), but exceeds the practical limits of inter-node through PCIe when RDMA is not available. In such cases, it is crucial that the prefill and decoding instances for a given request are handled by GPUs *within the same server node*, so communication is done via NVLink, as shown on the right side of Figure 7-17. On the contrary, the allocation on the left forces the massive KV cache to be transferred over the much slower inter-node network. This can create a severe communication bottleneck, effectively saturating the network fabric and throttling the overall system throughput.

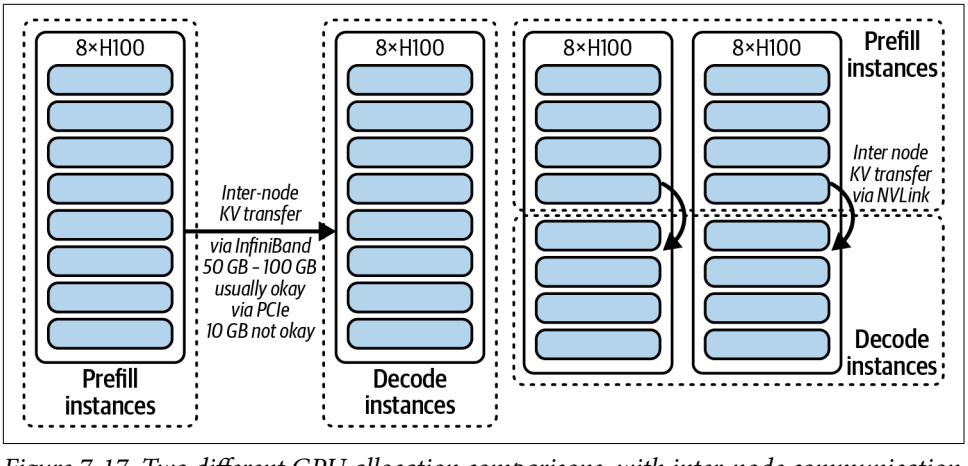

Figure 7-17. Two different GPU allocation comparisons, with inter-node communication on the left and intra-node communication on the right

Restricting intra-node placement of prefill instances and decode instances is not ideal. It forces you to use the same GPU hardware for both prefill and decode. It also limits model parallelism in many cases. For example, let's say one model instance is big enough that you want to use tensor parallelism with eight GPUs for prefill. In this case, inter-node communication would no longer be optional and would provide much more flexibility for scaling, especially when the prefill and decode workloads change. To combat the overhead of KV cache transfer in an inter-node setup, using RDMA across nodes is still a must, but there are several optimizations you can do to further reduce the overhead. For example:

- Instead of transferring the entire KV cache only when completely finished, you can chunk it to be transferred in smaller blocks, much like how streaming video works. You can tune the size of the blocks for the best performance.
- The transfers use asynchronous, nonblocking operations, so compute and communication can overlap as much as possible. When the prefill GPU is still performing the computation, the transfer is happening behind the scenes so the decode GPU can start its work sooner. This effectively hides the data transfer behind the stages of compute.
- Because the KV cache is per layer and is completely layer-local, it can also be transferred layer by layer. As the prefill stage computes the subsequent layers, the KV cache of the completed layers will be transferred, so the Decode stage can start on them.
- You can reduce or compress the KV cache's size.

With all these optimizations on KV cache transfer and improved scheduling, you can reduce the overhead of PD disaggregation to less than 1% of the overall per-request latency (Wang et al. 2025). Figure 7-18 shows a very simplified illustration of how KV cache transfer is hidden and overlaps with GPU compute so that, when managed correctly, it causes minimal overhead.

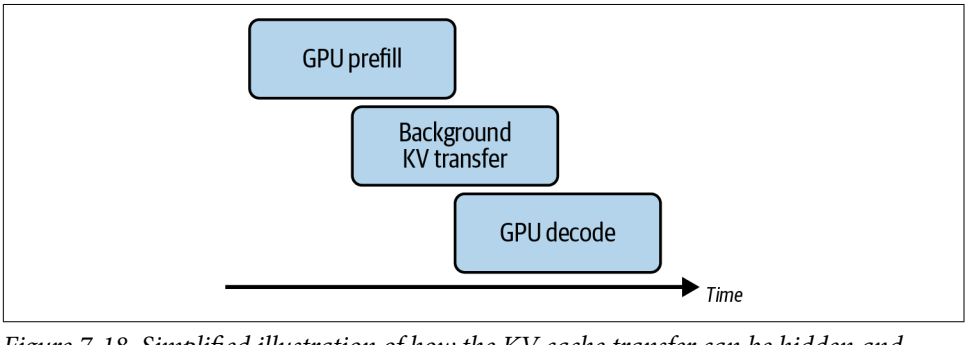

Figure 7-18. Simplified illustration of how the KV cache transfer can be hidden and overlap with compute

When to Use

With all the benefits of PD disaggregation, you might be wondering if you should use this technique by default. The answer is no. Figure 7-19 shows a detailed decision flowchart to help you decide which kind of serving to use based on your stage. But, simply put, use PD disaggregation only for large models under heavy load with advanced TTFT and ITL tuning needs, and stick with simpler setups for smaller models.

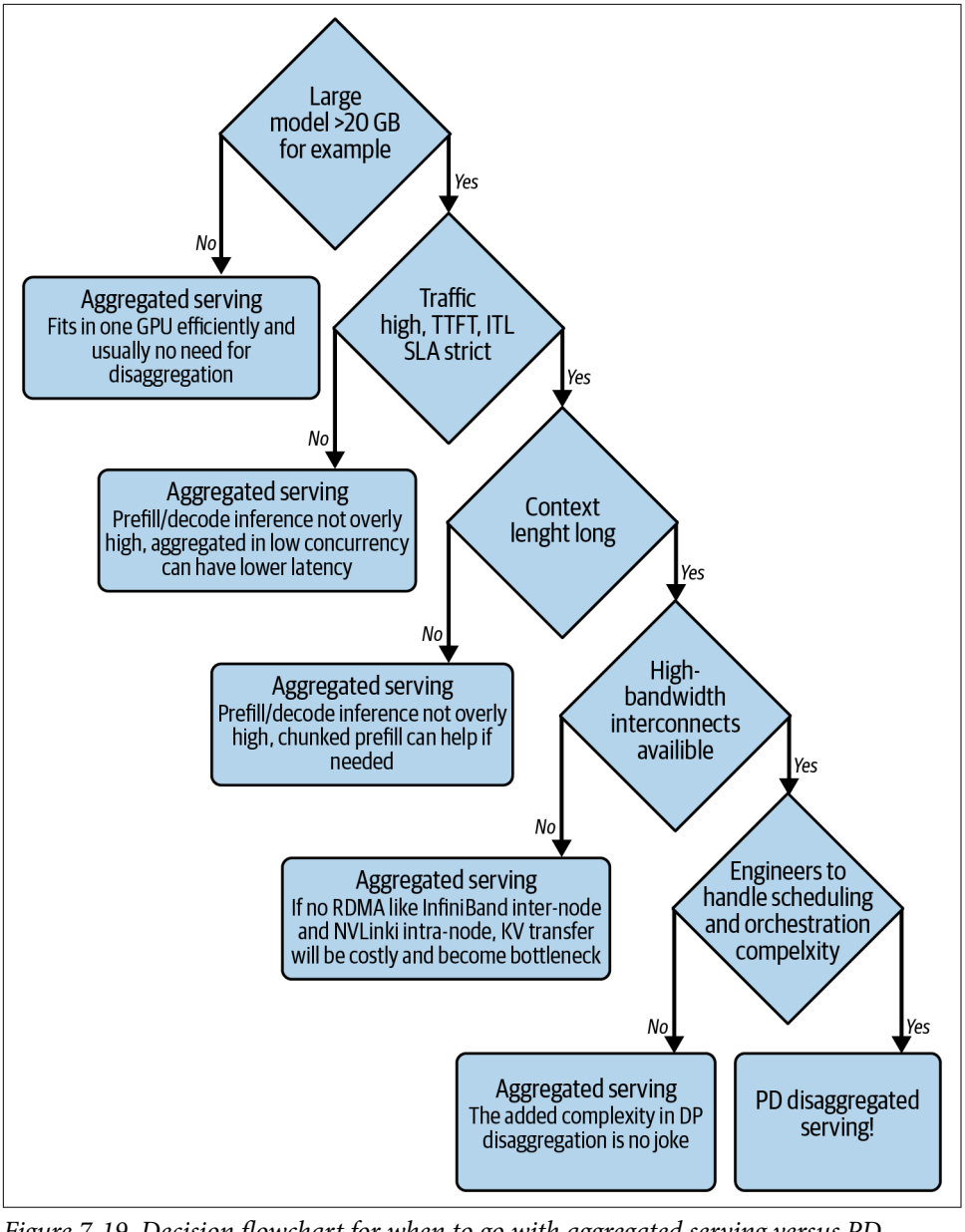

Figure 7-19. Decision flowchart for when to go with aggregated serving versus PD disaggregated serving

Advanced KV Caching

Up to this point, we have explored techniques that make LLM serving faster and more efficient: speculative decoding to shorten token-generation time, multiple parallelism strategies to utilize multiple GPUs for large models, and PD disaggregation to better balance the distinct demands of different stages of generation.

In real-world serving, we are always pushing the boundaries further, especially given modern applications and increasing demand for models that can reason over vast amounts of context to provide tailored, customized responses. For example, coding copilots must keep track of thousands of lines of source code so the model can generate changes in the right context. Conversational agents must sustain long interaction histories while drawing on tenant-specific knowledge bases to deliver consistent answers. Enterprise platforms must handle large document collections or transaction histories to extract insights, detect anomalies, or personalize decisions.

Long-Context Serving

Retrieval-augmented generation (RAG) is a very common strategy that retrieves the most relevant documents or records at query time and feeds them into the prompt, as you learned in Chapter 4. This approach keeps the context length and TTFT in the prefill stage manageable, while grounding responses in fresh, tenant-specific knowledge. RAG has proven invaluable in domains like customer support, enterprise search, and analytics, where accuracy heavily depends on dynamically retrieving the right slices of tenant-specific documents, records, or interaction histories. Cache-augmented generation (CAG, also discussed in Chapter 4) tries to cache most or even all of the relevant context as KV cache and treats it as reusable knowledge across different requests.

Prefix caching could be considered a naive implementation of CAG without advanced optimizations: you could treat all the relevant knowledge as static prefixes of the input and only append the dynamic prompts as suffixes to form the full input to send to the LLM. LLMs can leverage prefix caching to reuse the KV cache, achieving very low TTFT. The idea has been there for quite some time but hasn't been implemented and adopted as fast as RAG has, only picking up more recently.

There are shifts happening on two fronts that make long-context CAG-based serving more plausible. First, we're now seeing the rise of long-context models, with windows stretching from 100k to 1M tokens. This makes it possible to load most or all of the tenant's entire context directly into the model. In other words, rather than repeatedly relying on retrieval to inject information into each request, these models can *directly* attend to entire histories, large document sets, or transaction logs in a single extended context window. As these long-context LLMs become more and more capable, they are solving the "lost in the middle" problem, where LLMs are positionally biased to

emphasize information at the beginning and ending over that in the middle of the context.

On the engineering side, GPU memory is still very limited, despite newer chips increasing memory capacity. However, with efficient KV cache management techniques like offloading to CPU memory, SSD, or even remote storage, along with cross-replica routing, it is now very plausible to have an LLM cache large amounts of knowledge as KV cache to be reused when needed.

This shift reduces the complexity of managing retrieval pipelines, makes reasoning over large amounts of input smoother, and in many cases produces better responses. Without real-time retrieval, the long-context CAG approach can significantly reduce redundant computation, resulting in faster TTFTs than RAG achieves.

Cost and Latency Calculations

Figure 7-20 compares input tokens from RAG and long-context CAG serving. In this example, both have a system prompt with 500 tokens and can be easily cached in the KV cache to avoid recomputation across requests.

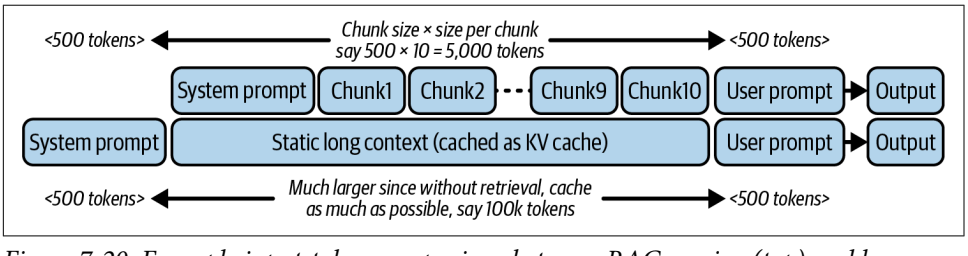

Figure 7-20. Example input-token comparison between RAG serving (top) and long-context CAG serving (bottom)

For RAG, depending on the user prompt, the model will dynamically generate the relevant knowledge chunks: in this case, 10 different chunks per 500 tokens. Because they vary by the user prompt, the KV cache hit rate will be very low. Adding the user prompt, LLM needs to run prefill non-cached on:

System prompt: 500 (cached)
RAG chunks: 500 × 10 = 5000
User prompt: 500
Total cached: 500
Total prefill regular non-cached: 5000 + 500 = 5500

With long-context CAG, both the system prompt and the static long context can be retrieved from the KV cache, so only the user prompt is dynamic. What we run prefill on is essentially:

System prompt: 500 (cached)
CAG context: 100,000
User prompt: 500
Total cached: 100,500
Total prefill regular non-cached: 500

Because CAG only takes in 1/10th of the input for prefill, it gains correspondingly in TTFT time. So if, with RAG, your TTFT is 5 seconds, long-context CAG could shave it down to around 0.5 seconds. In many production use cases, that's a night-and-day improvement. This doesn't even account for RAG embedding time and vector-search time prior to LLM generation, so the real-world TTFT savings would be even bigger.

However, you do need to account for the cost for caching long context, which will depend on whether the model is self-serve or uses an external vendor API. For example, Table 7-2 compares the costs for regular input versus cached input as we write this in late 2025. As you can see, the cached input is usually 10% to 25% of the regular input.

Table 7-2. Cost comparison of regular input and cached input across external vendors, in $ per 1M tokens

	GPT-5	Gemini 2.5 Pro	Claude Sonnet 4
Regular input	1.25	1.25	3
Cached input	0.125	0.31 (+ 4.5 storage/hr)	0.3 (+ 3.25 first write)

Let's include the GPT-5 costs in our example calculation. For RAG, that would be:

(500 cached input × $0.125 + 5500 regular input × $1.25)/10^6 = $0.007 per request

For long-context CAG:

(500 cached input × $0.125 + 100,500 regular input × $0.125)/10^6

= $0.013 per request

Long-context CAG is almost double the cost of RAG here! This is because we have 100,000 tokens of cached inputs rather than 5,000 tokens of RAG-based inputs. This calculation doesn't account for the RAG costs of offline indexing, storage, and online retrieval, but those usually cost much less than LLM calls, so in this hypothetical scenario, RAG will still be cheaper than CAG even though TTFT is a lot slower.

Self-Hosting LLMs

All of the preceding calculations assume that you are using a third-party API. Now we'll look at how to optimize a self-hosted model to make long-context CAG work faster and cheaper.

KV cache offloading

At this point, the KV cache is no longer just an implementation detail hidden inside the model—it needs to become a first-class citizen in the serving stack. It must be stored, scheduled, moved across devices, and evicted with the same care as model weights or input data. Systems such as LMCache are designed around exactly these aspects, elevating KV management into a core responsibility of large-scale LLM serving systems (see Figure 7-21).

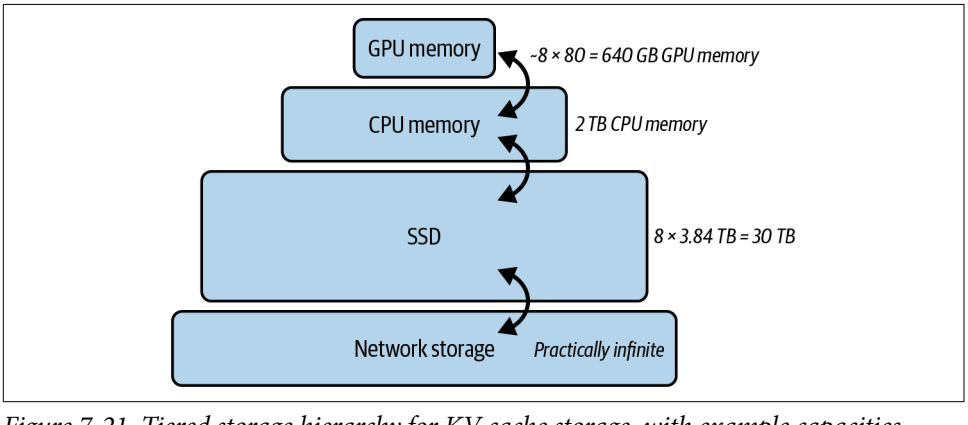

Figure 7-21. Tiered storage hierarchy for KV cache storage, with example capacities based on an AWS p5.48xlarge instance

The very first key feature that LMCache provides (as do a few other similar frameworks) is efficient tiered KV cache offloading. Figure 7-22 shows that offloading GPU memory to CPU memory can provide around triple the KV cache space. With SSD, you can get as much as 50 times more KV cache space! This means you can cache 50 times more long-context documents for many different uses or cache multiple tenants' documents within one instance. And all that extra space is literally free if you are already self-serving on the same instance. Adding distributed network storage, such as Redis or even S3, can be an option for more storage, should you need it.

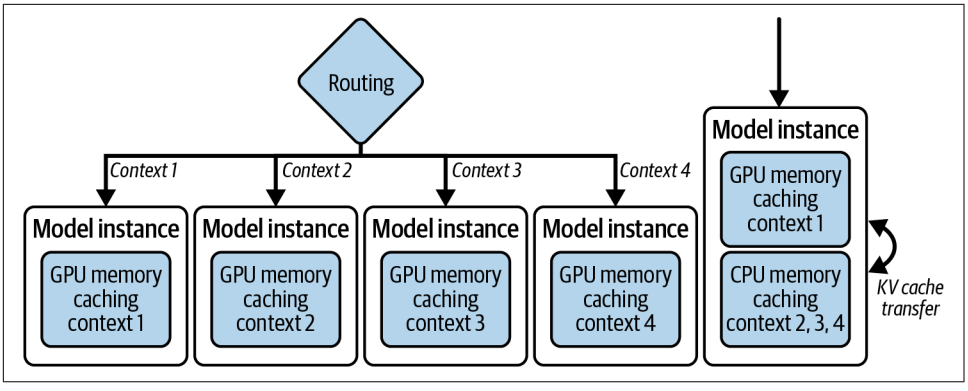

Figure 7-22. Long-context serving without KV cache offloading (left) and with KV cache offloading (right)

Figure 7-22 shows how KV cache offloading can help save costs over vanilla prefix caching. On the left, with prefix caching, no single model instance has enough GPU memory to hold more than one long-context KV cache. This forces us to create multiple model replicas in combination with a routing layer to ensure KV cache hits. On the right, with KV cache offloading, in this case only using CPU memory to triple the space, one model instance can keep all four long-context KV caches and, depending on the request prefixes, swap between GPU and CPU memory efficiently. If requests for one long context or one tenant cannot saturate a single model instance, this translates to 4x cost savings—millions of dollars per year.

KV cache compression

To further improve KV cache transfer performance and efficiency, especially when load is high and transfer bandwidth is limited, compressing the KV cache compression can really help. You can do this using traditional KV cache quantization or the more advanced CacheGen by LMCache, which encodes the KV cache into more compact bitstream representations based on its distribution, to ensure high quality and reduce size significantly. The smaller KV cache can be transferred much more quickly, with fewer network delays, and without too much GPU computation overhead during decompression. Combined with offloading, this also means more space in GPU memory to handle higher batches, increasing throughput and caching more context per model instance.

KV cache blending

Even all these KV cache management techniques don't mean that RAG is obsolete. Real-world enterprise data can be huge, highly dynamic, and often unbounded in scope. RAG still has the advantage in accessing a larger knowledge base, retrieving fresher information without data staleness, and potentially being easier to explain and trace for citations.

One novel, RAG-specific KV cache management idea is CacheBlend, supported by LMCache. CacheBlend lets you merge KV caches of different RAG chunks together, to reduce the amount of recomputation of all RAG chunks from scratch. Recall that LLMs can only match prefixes; different RAG chunks, or the same RAG chunks in a different order, can cause the KV cache to miss and do a full recomputation. You might think this could be solved by simply concatenating different KV caches, but it's not that straightforward. Self-attention doesn't just attend within each cache independently—it also learns cross-token interactions *between* caches. A naive concatenation would discard these relationships, effectively breaking the contextual dependencies that the model could have learned. Figure 7-23 provides an example where concatenating the two queries together works flawlessly in (b), whereas concatenating the KV caches does not in (c).

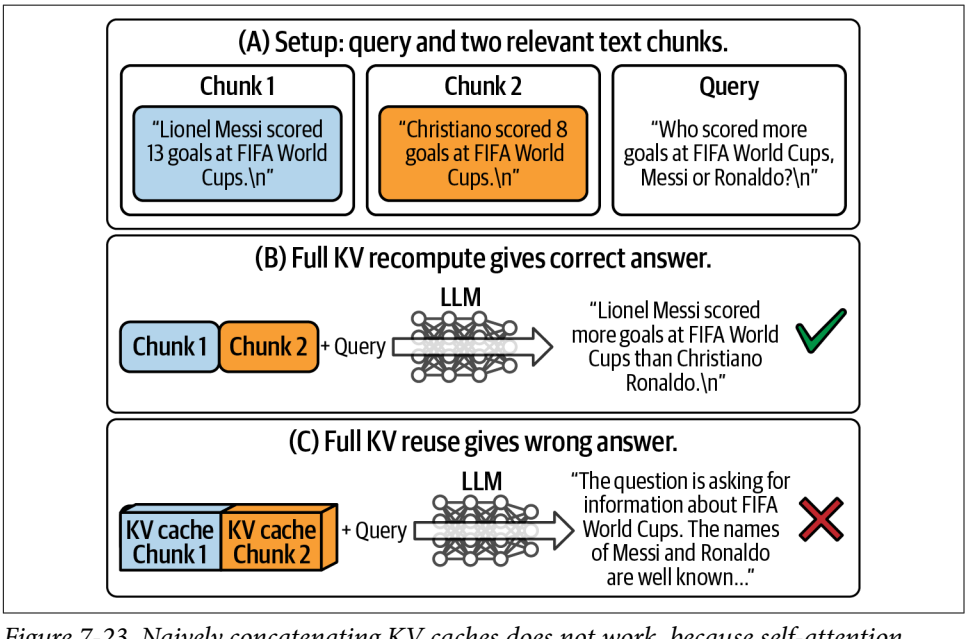

Figure 7-23. Naively concatenating KV caches does not work, because self-attention requires the later tokens to attend to the entire prior history (adapted from Yao et al., 2025)

To combat this issue and still leverage existing non-prefix KV caches, CacheBlend selectively recomputes a small subset of KV cache (the default is 15%, which can be tuned with `blend_recompute_ratios`) to preserve cross-token relationships. This avoids incurring the cost of full recomputation while achieving a very minimal drop in quality.

Hands-on LMCache

In this section, we'll show you some examples of how to enable LMCache in practice.

In the Google Colab notebook that accompanies this book, you can find the code that hosts this Qwen/Qwen3-14B model in two different configurations: vLLM serving with LMCache enabled for local CPU-based offloading, and vanilla vLLM serving with default prefix caching.

Enable LMCache and CPU offloading

First, let's look at an example command to enable LMCache with a vLLM server. Here, we enabled local CPU memory for KV cache offloading and set the CPU memory space to 150 GB:

```
export LMCACHE_USE_EXPERIMENTAL=True
export LMCACHE_LOCAL_CPU=True
export LMCACHE_MAX_LOCAL_CPU_SIZE=150.0

# Uncomment to test SSD offloading
# export LMCACHE_CHUNK_SIZE=2048
# export LMCACHE_LOCAL_DISK="local_kv_cache/"
# export LMCACHE_MAX_LOCAL_DISK_SIZE=120.0

# Start the vllm server
nohup vllm serve \
  Qwen/Qwen3-14B \
  --disable-log-requests \
  --kv-transfer-config '{
     "kv_connector": "LMCacheConnectorV1",
     "kv_role": "kv_both"
   }' \
   --max-model-len 40960 \
   --gpu-memory-utilization 0.9 \
> vllm.log 2>&1 &
```

Run benchmarks

In the example notebook, we ran two different benchmarks to show the performance difference between vanilla vLLM and vLLM with LMCache enabled. First, we made 30 sequential requests, each with unique prefixes. Then, we called those 30 sequential requests in *reverse order* to show the difference in KV cache space limit between LMCache with CPU offload enabled and vanilla vLLM using GPU memory only. Because the KV cache eviction setting defaults to least recently used (LRU) and KV cache space is limited, if we send new requests, we will not get the prefix cache hits anymore and latency will go up.

In the second benchmark, we use the vLLM bench CLI to submit 20 different long-context requests in three waves. First, we start with everything cold and send 20 requests sequentially, so neither the LMCache setup nor the vanilla vLLM has any KV cache. In the second wave, we send the same 20 requests again, sequentially. Last, we send the same 20 requests, but with 5 concurrent requests at a time, which helps us to analyze the performance change under higher load.

Performance analysis

As Figure 7-24 shows, when there are zero KV cache hits during the first run, LMCache is always slower than vanilla vLLM. This is likely due to the overhead of KV cache handling. This means that unless you expect your workload to have a lot of KV cache hits, vLLM without LMCache performs better and should be the default setup when prefixes are short or requests are very random.

However, during the second run, after warming up, as shown in Figure 7-25, for the first nine requests, which are all within GPU memory for both setups, they are neck and neck. After the tenth request, vanilla vLLM starts to reperform prefill computations, and latency shoots back up to the level of cold-processing latency. This happens because the older cache is all evicted while LMCache starts to retrieve the cache from CPU memory. The latencies when using CPU memory are not as good as when retrieving directly from GPU, but still much lower than performing full prefill computations (1 second versus 6–7 seconds).

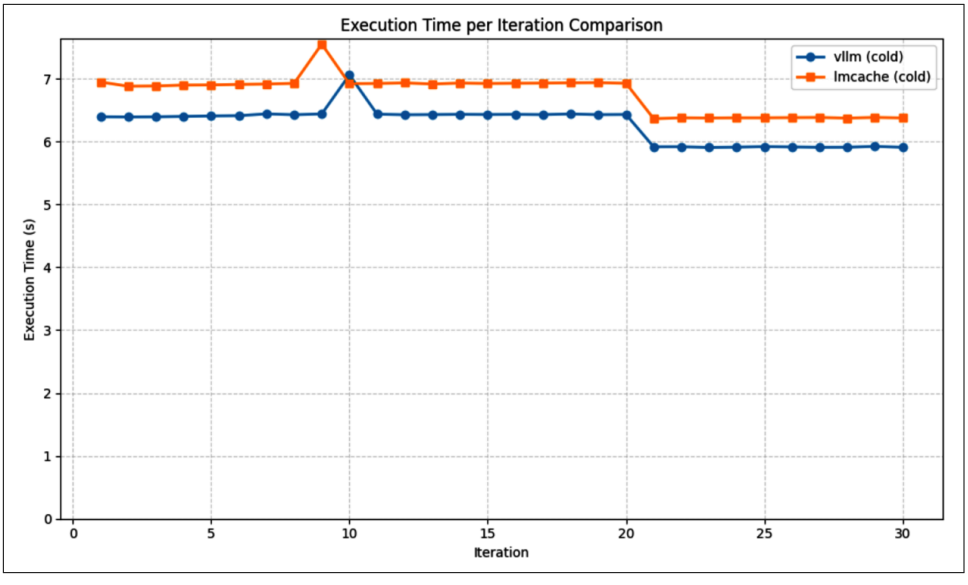

Figure 7-24. Execution time for 30 sequential requests sent to vanilla vLLM and LMCache with CPU offloading enabled during cold start without KV cache hit

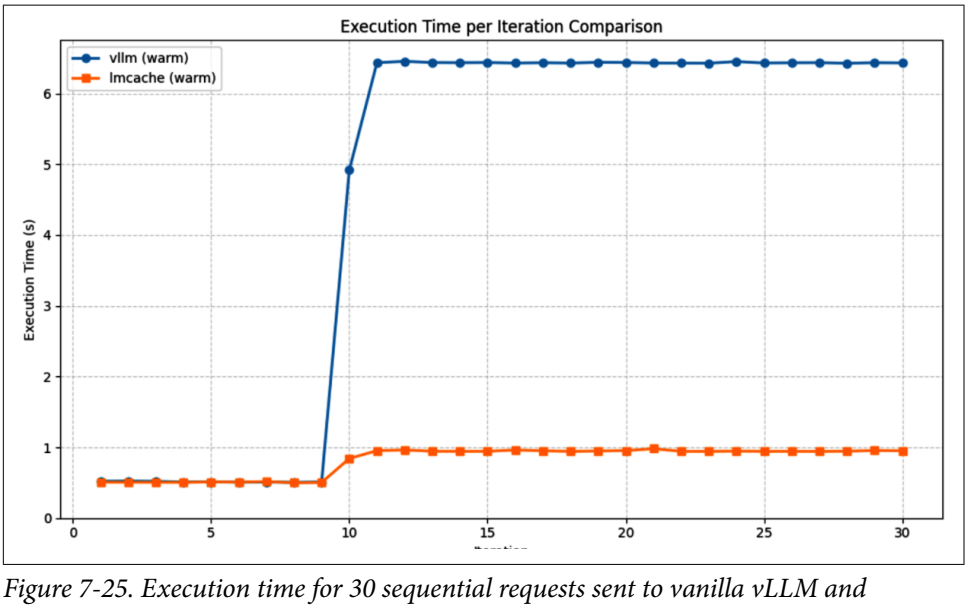

Figure 7-25. Execution time for 30 sequential requests sent to vanilla vLLM and LMCache with CPU offloading enabled and KV cache stored

During the second benchmark (Figure 7-26), we use the vLLM benchmark CLI to show how the performance changes under load. Because we are sending 20 different prefixes, vLLM can no longer store them all, making its median TTFT a lot higher than LMCache's. The gap is even bigger once we put in more concurrent requests: the vanilla vLLM setup gets bottlenecked by the KV cache space, because the same GPU memory needs to handle the KV cache and activations of five concurrent requests, which further squeezes the number of KV caches that can be stored. This is even more apparent in the throughput metrics (Figure 7-27), where LMCache easily handles the increased load while vLLM is bottlenecked at around 4,700 tokens per second—around 16 times less than the throughput of the LMCache setup.

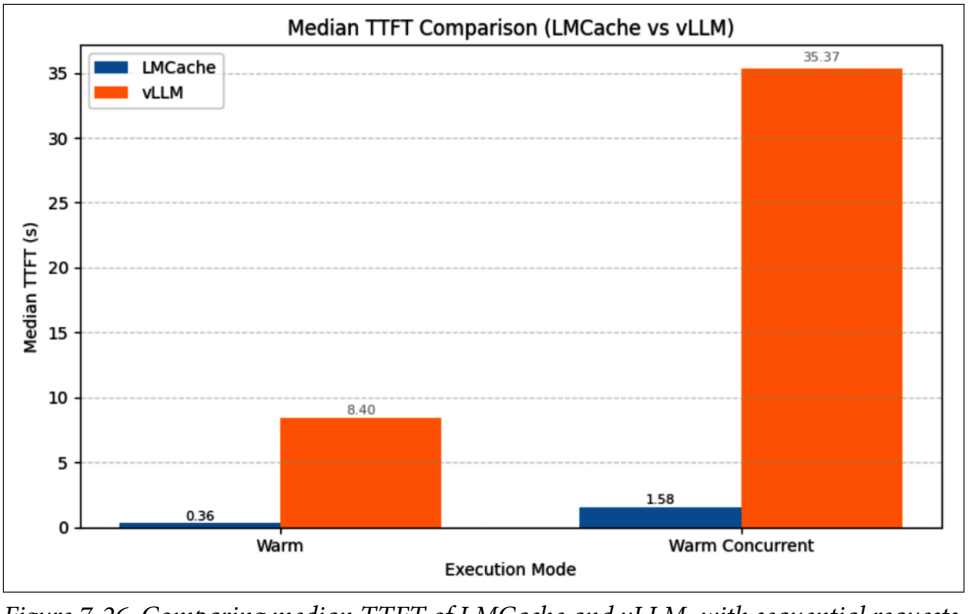

Figure 7-26. Comparing median TTFT of LMCache and vLLM, with sequential requests (left) and five concurrent requests (right)

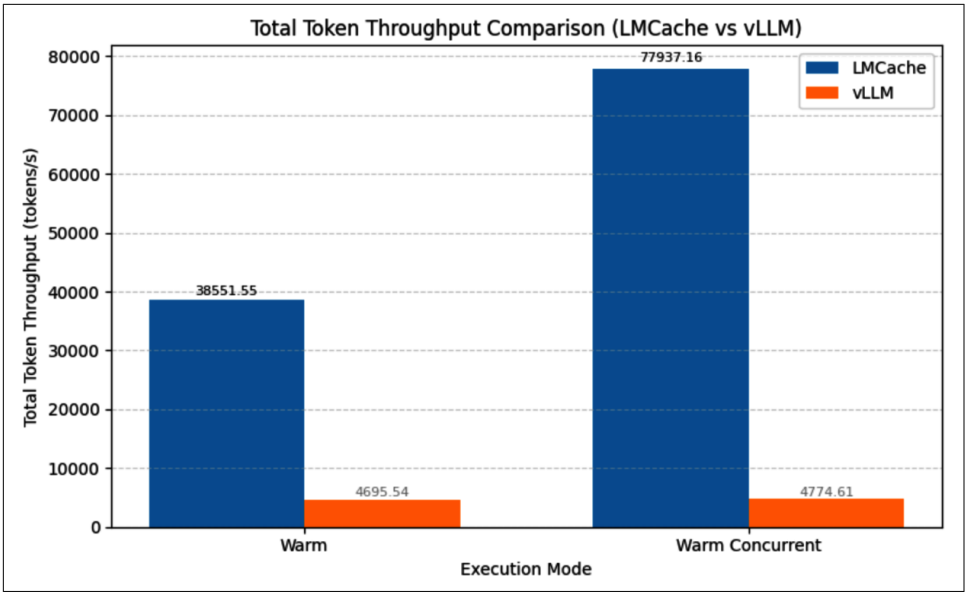

Figure 7-27. Comparing throughput with LMCache and vLLM, with sequential requests (left) and five concurrent requests (right)

Summary

In this chapter, we explored advanced techniques for large-scale LLM serving and optimization. We began with speculative decoding, a powerful strategy to reduce latency by predicting multiple tokens in parallel and verifying them efficiently. We then examined different model parallelism techniques and combinations such as tensor parallelism (TP) and pipeline parallelism (PP), which enable scaling large models across multiple GPUs by splitting computations along dimensions or sequential layers. Building on that, we discussed prefill-decode (PD) disaggregation , which decouples the prefill and decode stages so each can run on the most suitable hardware and configurations, enabling finer-grained scaling and better hardware utilization. Finally, we examined advanced cache-management techniques such as offloading KV caches across memory and storage tiers (GPU, CPU, SSD) to balance speed and capacity, cache compression methods that shrink memory footprint with minimal accuracy loss, and novel KV cache-blending techniques to merge non-prefix KV caches for RAG usage.

Modern LLM serving has evolved far beyond running a single model instance efficiently. Today it is a hierarchical stack. At the lowest level lies the highly optimized CUDA and Triton kernels layer, which maximizes GPU compute and memory utilization efficiency. Above that, the execution-engine layer (containing, for instance, vLLM, SGLang, or TensorRT-LLM) handles token scheduling, batching, streaming,

and coordinating prefill and decode execution across GPUs. The cache-management layer (such as vLLM's KV manager and LMCache) maintains KV cache allocation, compression, and tiered offloading across GPU, CPU, SSD and distributed storage to balance latency and capacity. Above it all, the routing and orchestration layer such as NVIDIA's Dynamo (Figure 7-28) and llm-d operates to direct each request to a prefill or decode worker and to the optimal model instance, based on real-time signals such as KV cache locality, queue depth, and latency budgets. It also governs global resource management and autoscaling, using aggregated utilization and latency metrics to optimize throughput and cost efficiency across the entire fleet.

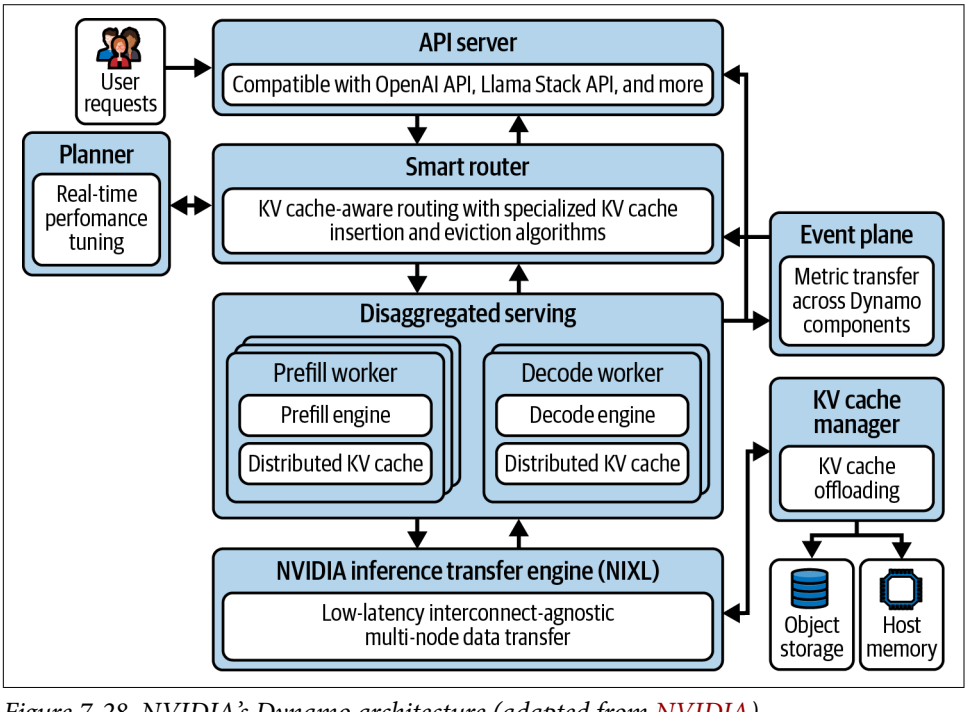

Figure 7-28. NVIDIA's Dynamo architecture (adapted from NVIDIA)

Together, all these layers form an integrated optimization stack to deliver state-of-the-art latency, throughput, and cost efficiency. Achieving production-grade performance requires tuning across all layers in concert, since bottlenecks at any single stage can erase gains elsewhere. For organizations operating LLMs at scale, such system-wide efficiency is not just a technical pursuit: it can be both a use-case enabler and a core economic differentiator.

This concludes our three-chapter focus on LLM optimization techniques. Across these chapters, we've examined why and where serving complexity arises, explored the fundamental techniques that address it, and then advanced to multilayered optimizations that unify compute, memory, and orchestration. Together, they provide the foundation for designing LLM systems that are scalable, performant, and cost-efficient in real-world production environments.

In the next chapter, we'll look closely at different serving frameworks to show you how a highly optimized LLM serving stack such as vLLM works under the hood.

CHAPTER 8

LLM Serving Frameworks

Over the previous chapters, we've explored the fundamentals of LLM serving—system design, service implementation, and practical optimization techniques. This chapter shifts to the foundation layer—the serving frameworks that implement and execute model inference with different optimization techniques under real production constraints. We'll discuss four widely adopted open source serving frameworks you're likely to encounter in the wild: vLLM, TensorRT-LLM, SGLang, and llama.cpp. Each has a distinct philosophy, hardware footprint, and battle-tested technology, and is backed by active communities and growing production usage.

Because it's the most broadly applied framework, we'll take a deep dive into vLLM—its architecture, initialization and model-execution process, request and token-level scheduling, and layered optimization strategy. Understanding vLLM's internals will give you strong intuition for how LLM frameworks work in practice and make it easier to evaluate the trade-offs in other frameworks.

Next, we'll cover the remaining frameworks with concise, decision-oriented overviews and short examples. We'll close the chapter with the evaluation method we use to compare serving frameworks.

After reading this chapter, you'll have a solid grasp of what LLM serving frameworks are, why we need them, how they work under the hood, and how to evaluate them for your use case. In the next chapter, we'll put the optimization techniques and serving framework into practice—tuning LLM performance with the vLLM serving framework.

Why We Need Specialized LLM Serving Frameworks

Before the LLM era, there were already many general-purpose model serving frameworks, such as TensorFlow Serving, TorchServe, and even generalized inference

platforms like NVIDIA Triton. These frameworks and platforms were originally designed for deep-learning workloads like image recognition and structured data inference. Those workloads usually have short input sizes, fixed-shaped tensors, and predictable latency requirements, and their primary optimization is usually batch processing.

At this point in the book, it should be clear that serving LLMs is fundamentally different from serving traditional machine-learning models, such as image classifiers or recommendation models. LLM serving and optimization introduce a new set of challenges, including:

Autoregressive generation
: LLMs generate outputs one token at a time. Unlike with image models, inference sessions may remain open for seconds or minutes.

Context length explosion
: Models must handle input prompts of sizes ranging from a few tokens to hundreds of thousands or even a million tokens. KV cache memory management becomes a critical bottleneck.

Continuous batching
: Input and output lengths vary wildly across requests. Static batching strategies underutilize GPUs.

Streaming requirements
: Users expect time to first token (TTFT) within hundreds of milliseconds, as well as continuous token streaming.

Resource utilization
: GPUs are expensive. Wasting GPU FLOPS due to fragmentation or idle tokens is unacceptable at scale.

To meet these needs, a new class of frameworks has emerged—specialized LLM serving frameworks, such as vLLM, TensorRT-LLM, and SGLang. These frameworks introduce innovations such as paged KV caching, continuous batching, LLM-specific quantization, and speculative decoding (discussed in previous chapters) to address the unique challenges of LLMs. With the help of these LLM serving frameworks, we can squeeze more throughput and lower latency from modern accelerators and deliver far better efficiency—which is why they have become the mainstream choice for LLM serving.

vLLM

vLLM is one of the most popular open source frameworks for LLM serving. Its adoption has been rapid across both open source communities and enterprise teams

because it solves the pain points of running LLMs at scale: long prompts, high memory demands, and the need to serve many users simultaneously.

At its core, vLLM introduces paged KV caching and incorporates it with continuous batching, two innovations that dramatically improve throughput and reduce latency. It also supports features like quantization, speculative decoding, streaming responses, and multi-GPU and distributed execution. These capabilities suit it to a wide range of scenarios—like interactive chatbots and RAG systems, batch text generation, multi-tenant serving, and real-time applications.

Many people like vLLM because it "just works." It's battle-tested, it's easy to integrate with open source models and self-fine-tuned models, it maximizes GPU efficiency without requiring deep system-level tuning, and it delivers predictable performance gains out of the box. This mix of usability and efficiency is why vLLM has become a go-to choice for LLM serving in production.

Another strength of vLLM lies in its architecture. Its clean, extensible design makes it easy to incorporate the latest academic advances and practical optimizations. Combined with its large, active community, vLLM is not only well suited to today's serving challenges but is also positioned to adapt and thrive as future needs evolve.

In this section, we'll take a deep dive into vLLM—starting with its high-level system design, then walking through the model initialization workflow and request-handling pipeline. From there, we'll move into more advanced topics, such as how generation requests are prioritized and optimized at the framework level to support a wide range of LLMs.

vLLM's Architecture

vLLM is optimized for a *single model service setup*, in which each instance initializes one model at startup. It offers two primary ways to use that model:

`LLM` *class (in-process library)*
: A pure-Python local interface for offline inference—no separate server or web API required. This "library mode" is ideal when you want to plug vLLM directly into an existing service or batch workflow.

API Server (OpenAI-compatible)
: A standalone HTTP server for multiclient and production use and streaming, with easy integration via the Chat/Completions API.

Using the `LLM` class gives you tight control, low overhead, and easy integration with your own serving stacks, while the API server provides a ready-to-use network endpoint for broader deployment. Let's see the LLM class example first:

```
# initialize LLM model
llm = LLM(
```

```
    model="Qwen/Qwen3-7B-Instruct",
    trust_remote_code=True, # Qwen uses custom modeling code
    dtype="float16",  # Use float16 for GPU
    max_model_len=32768,
    gpu_memory_utilization=0.8,
)

# run model generation requests
outputs = llm.generate(prompts, sampling)
```

Start the vLLM API server with the `vllm serve` command:

```
# start vLLM API server
vllm serve Qwen/Qwen3-7B-Instruct \
  --trust-remote-code \
  --dtype bfloat16 \
  --max-model-len 32768 \
  --gpu-memory-utilization 0.8

# call the Qwen model via the OpenAI-compatible API
curl http://localhost:8000/v1/chat/completions \
 -H "Content-Type: application/json" \
 -d '{
   "model": "Qwen/Qwen3-7B-Instruct",
   "temperature": 0.7,
   "max_tokens": 256,
   "stream": true
 }'
```

Now let's turn to vLLM's major internal components, as shown in Figure 8-1.

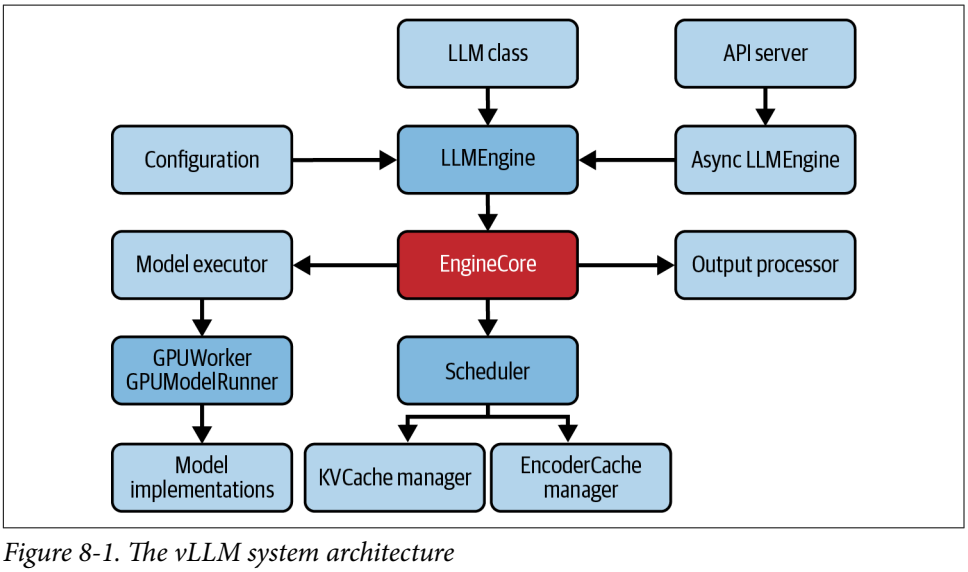

Figure 8-1. The vLLM system architecture

LLMEngine and EngineCore

`LLMEngine` is the high-level interface and main entry point for vLLM's inference system. It serves as the public API that users interact with, while internally coordinating all the underlying components.

`LLMEngine` integrates all components into a cohesive system with clear data flow. It handles both synchronous and asynchronous serving scenarios and manages the overall request lifecycle, such as orchestrating the request-processing pipeline and managing the request queue and configuration.

`EngineCore` is the central orchestrator of vLLM's inference engine. It integrates the model executor, output processor, and scheduler, and serves as the "inner loop" that coordinates all major components and manages the complete request-processing pipeline.

Scheduler

The `Scheduler` component acts as the "traffic controller" for the entire inference pipeline. It manages the compute resources and orchestrates token calculation across requests. Its primary responsibility is to allocate limited computational resources (such as GPU memory, KV cache blocks, and processing capacity) efficiently among competing requests while maximizing throughput and maintaining fair access.

From an optimization perspective, `Scheduler` is responsible for systemwide, model-agnostic optimization strategies such as token-level scheduling, dynamic batching, prefix caching, and chunked prefill. In contrast, model-specific optimizations are applied within the `ModelExecutor` and `GPUWorker`, which we'll discuss next.

`Scheduler` encapsulates its model-execution plan into a comprehensive data structure called `SchedulerOutput`, which serves as the "work order" from the scheduler to the `ModelExecutor`, containing all the information needed to execute a batch of requests. The `ModelExecutor` responds to run the model. `SchedulerOutput` essentially tells the `ModelExecutor`: "Here's a batch of requests to process. Here's how many tokens each one should get, here are their input data and parameters, here are the memory blocks allocated for them, and here are any special processing requirements."

The `ModelExecutor` uses the information in `SchedulerOutput` to prepare the actual GPU batch (with flattened input IDs, attention metadata, and KV cache blocks) and execute the model forward pass, then return the results back to the scheduler for the next iteration.

ModelExecutor, (GPU) Worker, and ModelRunner

Since vLLM hosts and runs each model in its own separate process or process group, it employs a layered architecture to handle the cross-process communication, orchestrate distributed worker groups, and handle various model forward-pass execution details. This architecture includes three components:

`ModelExecutor`
: Orchestrates and manages multiple worker processes

`GPUWorker`
: Works as the worker interface that runs in each worker process and manages device/model lifecycle

`GPUModelRunner`
: Actually executes and runs the neural network

This separation of concerns allows each component to focus on its specific model-execution responsibility while maintaining clean interfaces between them. Figure 8-2 illustrates this layered cross-process model-execution architecture.

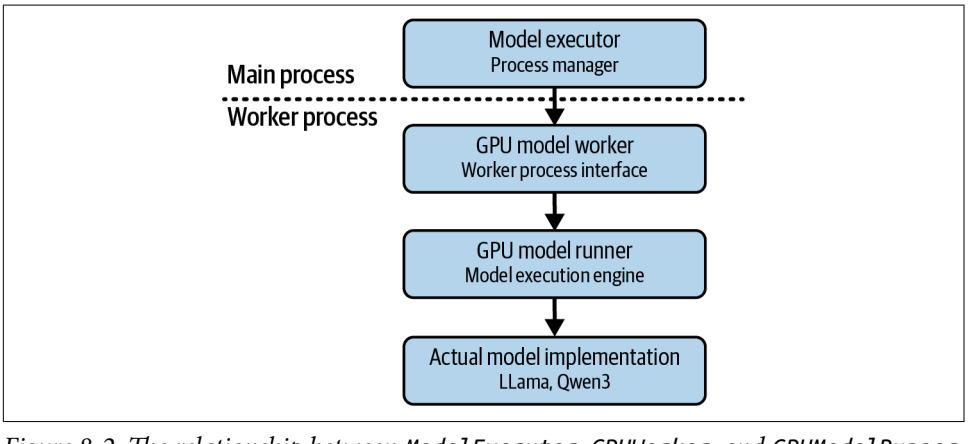

Figure 8-2. The relationship between `ModelExecutor`, `GPUWorker`, and `GPUModelRunner`

With this architecture in mind, let's now explore how these components come together in practice—from model initialization to request execution.

Model Initialization Workflow (with Multi-Process Worker)

In this section, we'll walk through the model initialization workflow in vLLM, covering how it brings up internal components, sets up the worker process group, loads the model from its source into GPU memory, and establishes the communication links.

Because vLLM supports a wide range of models and multiple execution styles—single-device serving, multi-device serving on one node, and even multi-node clusters—the initialization logic can feel complex. To make it easier to follow, we'll focus on a practical example: initializing an LLM with a multi-process setup, the configuration most commonly used in production today. See Figure 8-3 for a visualization of this process.

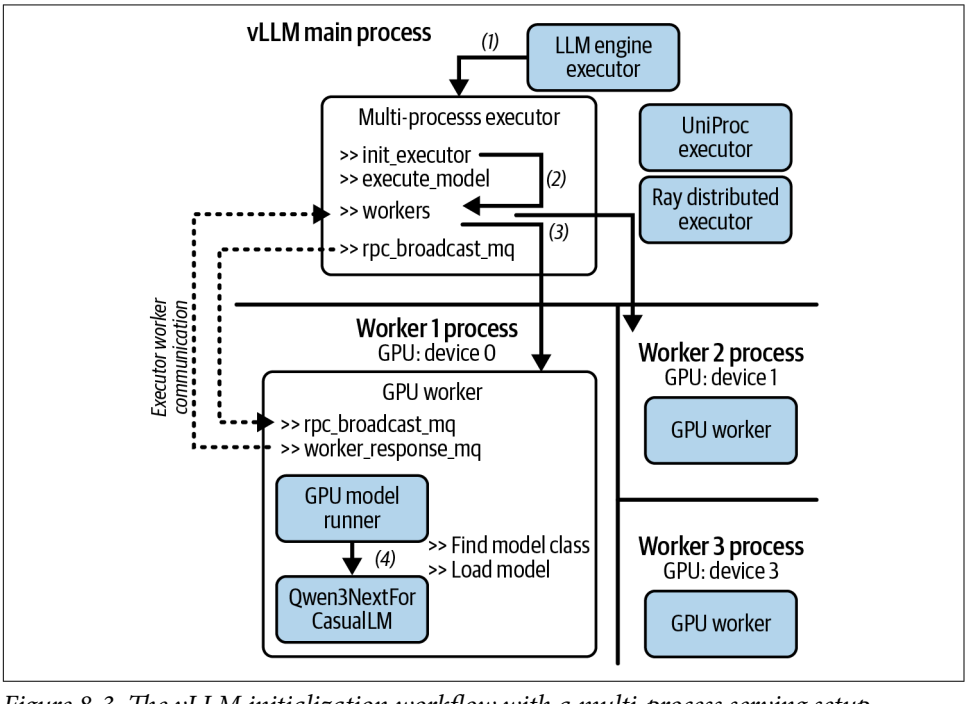

Figure 8-3. The vLLM initialization workflow with a multi-process serving setup

Let's walk through the four steps shown in Figure 8-3:

1. *Initialize all components in the vLLM main process*

   When creating an `LLM()` instance, we pass in configurations that define the model-setup, resource-usage, and serving-optimization parameters. In the following example, we configure vLLM to host a Qwen model from Hugging Face using a multi-process group with four worker processes. We used a multi-process (MP) backend for single-node model deployment with multiple GPUs, but you can also choose the "ray" backend for cross-machine setup:

```
llm = LLM(
  model="Qwen/Qwen2.5-7B-Instruct",
  # specify 4 workers
  tensor_parallel_size=4,
  # use multi-process model executor
```

```
        distributed_executor_backend="mp"
    )
```

The `LLM` class initializes all major components within the vLLM main process and distributes the configuration to the appropriate modules—including the `LLMEngine`, `Scheduler`, `KVCacheManager`, and `MultiProcessExecutor`.

2. *Create a worker process group*
   During initialization, the `MultiProcessExecutor` spawns four worker processes for distributed model execution. It also sets up an `rpc_broadcast_mq` message queue to pass signals and commands to these workers.

3. *Initialize the worker processes*
   Each worker process runs a `GPUWorker`, which is responsible for executing model inference and communicating with the `MultiProcessExecutor`. The `GPUWorker` sets up the CUDA device, establishes cross-process communication, and loads and initializes the model within the worker process. For example, the `GPUWorker` maintains a message queue (`worker_response_mq`) to send inference results back to the `ModelExecutor`.

4. *Prepare and load the model*
   The `GPUModelRunner` is responsible for locating the correct model implementation based on the configured model name. In our example, it looks up the Qwen model in vLLM's internal model registry and selects the `Qwen3NextForCausalLM` implementation. It then calls the class's `__init__` function to load the model weights onto the GPU.

Generation-Request Execution Workflow

With the model loaded, initialized, and set up in vLLM, it's ready to serve. Let's look at a high-level workflow for executing generation requests (pictured in four steps in Figure 8-4).

When you run `llm.generate(prompts, sampling_params)`, internally, here is what vLLM does:

1. The `Processor` component validates and preprocess the raw inputs into a `Request` object, including tokenizing the input prompts.
2. `LLMEngine` runs an execution loop, which calls the `EngineCore` repeatedly to process requests. The `EngineCore` lets the `Scheduler` decide the next batch of requests to run and the tokens to be processed. `Scheduler` also applies many optimizations in this step, such as paged attention and continuous batching.
3. `EngineCore` passes the scheduled token (`SchedulerOutput`) from `Scheduler` to `MultiProcessExecutor`.

4. `MultiProcessExecutor` passes the request to the worker process and delegates the GPU worker to run the model forward pass to calculate the given tokens. This is where the actual model execution happens.
5. `EngineCore` passes the model output to the output processor, which creates the final response output to the customer.

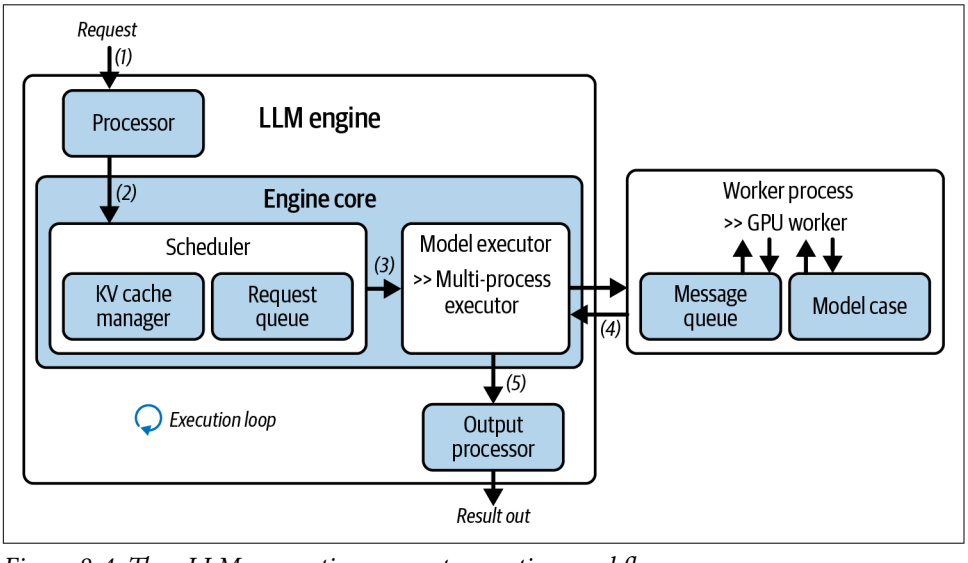

Figure 8-4. The vLLM generation-request execution workflow

From this serving workflow, you can see that the actual model execution and model-specific optimization are handled by the `GPUWorker`, while the general LLM-serving planning and optimization are orchestrated by the `Scheduler`. Next, let's take a deeper look at how the `Scheduler` works.

Scheduler Deep Dive

In general, the `Scheduler` functions as the central traffic controller for executing generation requests in vLLM's inference pipeline. We will first examine the key considerations that shape the vLLM `Scheduler`, and then dive into its request scheduling workflow:

Request resource orchestration

The `Scheduler` orchestrates the entire lifecycle of a request, from its arrival to completion. It receives incoming requests, organizes them into separate `WAITING` and `RUNNING` queues, and makes dynamic decisions about which requests to dispatch based on available computational resources such as GPU memory, KV cache blocks, and token budgets.

The `Scheduler` also tackles complex resource-allocation challenges, including speculative decoding, prefix caching, and multimodal inputs. Its goal is to maximize throughput while maintaining fairness across concurrent requests, ensuring an efficient and balanced use of system resources.

Token-level resource allocation and scheduling
: One of vLLM's key strengths is that it uses a unified token-based scheduling approach, rather than separating the prefill and decode phases. Unlike a request-based scheduler, which treats each request as a whole unit, vLLM's `Scheduler` operates on a token-by-token basis. At each scheduling step, it decides how many tokens each request can process, while respecting global constraints such as maximum batch size and GPU memory limits.

  Compared to a request-based scheduling strategy, token-based scheduling offers finer-grained control. It balances competing demands across requests by allocating token budgets, managing KV cache blocks for attention states, and supporting encoder inputs in multimodal models. This approach ensures the total computational load stays within system limits while still maximizing opportunities for parallel execution.

Optimization integration hub
: The `Scheduler` acts as the central integration point for model-agnostic performance optimizations. It coordinates techniques such as *prefix caching* (reusing previously computed states), *speculative decoding* (predicting future tokens), *chunked prefill* (efficiently processing long sequences), and *distributed KV cache transfers* (enabling multi-GPU execution).

  Crucially, the `Scheduler` makes adaptive decisions on when and how to apply these optimizations. Its choices are guided by request characteristics, resource availability, and overall system state, ensuring that each optimization improves performance without introducing conflicts or inefficiencies.

Dynamic load balancing
: The `Scheduler` monitors system resources and requests characteristics to make real-time decisions that balance latency and throughput. It adapts to dynamic events—arrivals, completions, preemptions, and resource limits—by reevaluating which requests to run, how many tokens to process, and when to apply optimizations. Through efficient queue management and resource allocation, it coordinates with workers to sustain optimal performance.

Request lifecycle management
: The `Scheduler` monitors each request throughout its lifecycle—from arrival in the `WAITING` queue through execution in the `RUNNING` queue to final completion.

The `Scheduler` also applies advanced scheduling policies, such as first-come-first-served (FCFS) or priority-based ordering, to determine execution order. When resources are limited, the `Scheduler` can preempt lower-priority requests, reallocating capacity to higher-priority tasks. It also manages smooth transitions between request states, ensuring both efficiency and fairness in execution.

Request scheduling workflow

The core task of the `Scheduler` is to determine the appropriate number of tokens for each incoming generation request and pass them to the model for a forward pass. To fully utilize model and hardware capacity while maintaining a balanced user experience, the `Scheduler` applies a variety of model-agnostic optimizations during this token selection process. The overall request-scheduling logic is illustrated in Figure 8-5.

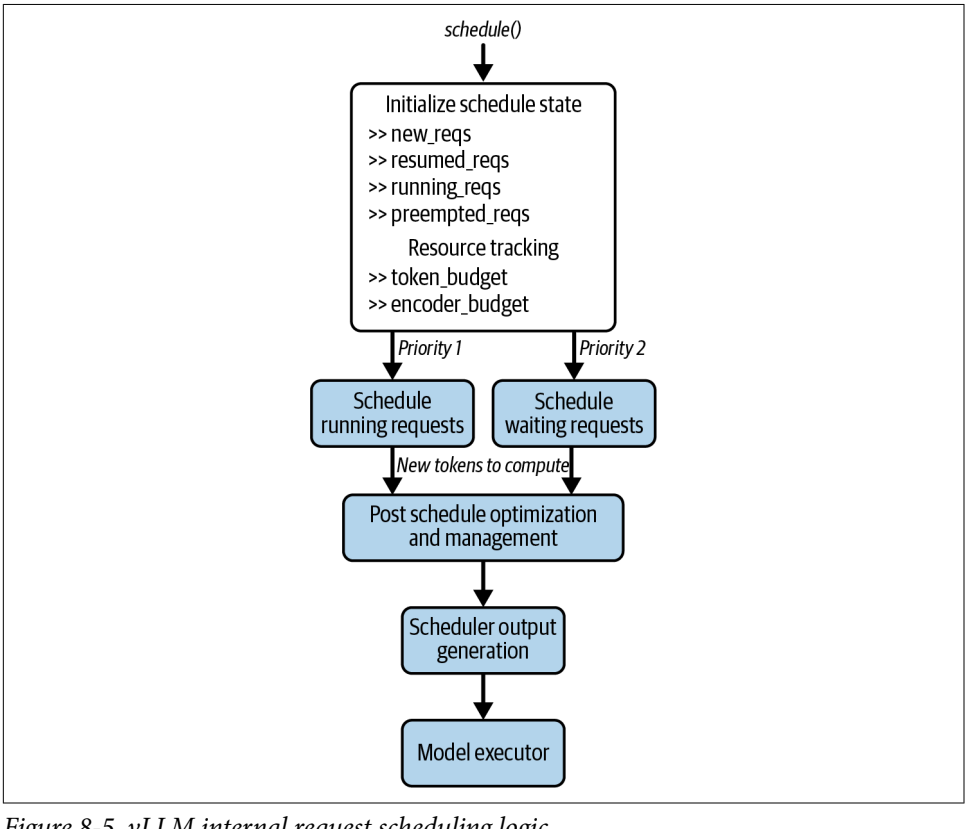

Figure 8-5. vLLM internal request scheduling logic

From Figure 8-5, you can see the vLLM scheduler begins its scheduling cycle by building an internal *schedule state*, which gathers all relevant requests: newly arrived requests, previously paused requests that need to be resumed, currently running requests, and any requests that were preempted earlier. During this initialization phase, the scheduler also updates its accounting of available resources, such as how many decoding tokens remain within the current token budget and how much encoder capacity is still available for multimodal workloads. This initial state frames all decisions that follow.

Once initialization is complete, the scheduler begins assigning compute in two prioritized waves. Running requests are handled first, because they already occupy KV cache blocks and have active generation timelines. For each running request, the scheduler determines how many new tokens must be generated, verifies encoder constraints when multimodal inputs are involved, and checks whether there is enough available KV cache to continue execution.

At this stage, vLLM applies several optimizations that significantly reduce latency and improve throughput. Chunked prefill allows partial processing of long prompts without requiring the entire prompt to be encoded in a single step. Prefix caching enables the scheduler to reuse KV cache blocks that were computed earlier for identical prefixes across different requests. Meanwhile, speculative decoding generates draft tokens ahead of time, enabling faster generation if those predictions later match the model's verified output. If at any point resources become constrained, the scheduler may preempt lower-priority work to ensure fairness and consistency.

After all running requests have been handled, the scheduler turns to the waiting queue. Any requests that can fit within the remaining token and encoder budgets are activated and incorporated into the current execution batch. Although waiting requests receive lower priority, they still benefit from the same optimizations applied to running requests whenever applicable.

When request assignment is complete, the scheduler enters a post-processing phase that performs additional coordination work necessary for the upcoming decoding step. In this phase, the scheduler tracks which LoRA adapters must be active for each request, prepares encoder inputs for multimodal tasks, and finalizes any draft-token predictions that will be used during speculative decoding. The goal of this stage is to consolidate all scheduling decisions into a clean, executable plan.

Finally, the scheduler assembles a `SchedulerOutput` object that summarizes the results of the entire cycle. This output lists the requests that are newly scheduled, the number of tokens allocated to each request, and the total number of scheduled tokens across the batch. It also includes metadata needed by the model executor, such as KV cache assignments, prepared encoder inputs, or multimodal routing information. The model executor then consumes this output to run the actual forward passes that produce new tokens.

In addition, to support a model-agnostic request–token scheduling process, the vLLM `Scheduler` operates at the token level. It iterates through each request—first from the running queue, then from the waiting queue—and determines how many tokens from which request to include in the next forward pass, while ensuring the total token count remains within the model's limits.

At the implementation level, the `Scheduler` seeks to minimize the gap between `num_computed_tokens` (the number of tokens already processed) and `num_tokens_with_spec` (the total tokens to be processed, including prompt, output, and speculative tokens). This goal of closing the gap drives the chain of optimization strategies applied during scheduling. Here's how that looks in the code:

```
while req_index < len(self.running) and token_budget > 0:
    request = self.running[req_index]
    num_new_tokens = (request.num_tokens_with_spec +
                      request.num_output_placeholders -
                      request.num_computed_tokens)
```

Separate Prioritization and Optimization

In vLLM scheduling, the request queues determine the order of request processing, while the comparison between `num_computed_tokens` and `num_tokens_with_spec` dictates how many tokens each request is allowed to execute.

This clear separation of request prioritization and token-level scheduling allows the `Scheduler` to combine diverse request-prioritization strategies with LLM execution optimizations in a unified framework, maintaining fairness across concurrent requests while ensuring efficient use of system resources.

Applying model optimization techniques

During scheduling, vLLM integrates several of the optimization techniques we introduced in Chapters 6 and 7, including chunked prefill, prefix caching, guided decoding (via grammar-constrained finite-state machines), PD disaggregation, and more. To illustrate how these work in practice, let's walk through two quick examples:

Chunked prefill

The `Scheduler` ensures that the number of new tokens executed for a request does not exceed the configured prefill chunk size—`scheduler_config.long_prefill_token_threshold`:

```
if (0 < self.scheduler_config.long_prefill_token_threshold \
                  < num_new_tokens):
   num_new_tokens = self.scheduler_config.long_prefill_token_threshold
```

Prefix caching
: The `Scheduler` handles prefix caching to reuse computed KV cache blocks from local or remote cache:

```
# Get already-cached tokens.
if request.num_computed_tokens == 0:
    # Get locally cached tokens.
    new_computed_blocks, num_new_local_computed_tokens = \
          self.kv_cache_manager.get_computed_blocks(
                              request)

    # Get externally cached tokens.
    if self.connector is not None:
        num_external_computed_tokens, load_kv_async = (
            self.connector.get_num_new_matched_tokens(
                request, num_new_local_computed_tokens))
```

We encourage you to explore other optimization techniques and their implementations—such as guided decoding and prefill-decoding—further on your own. A good starting point is the *Scheduler.py* code implementation. Additionally, Aleksa Gordić's blog post "Inside vLLM: Anatomy of a High-Throughput LLM Inference System" is an excellent resource.

vLLM's Layered Optimization Strategy

One of the most important optimization philosophies in vLLM is that *optimization must happen at the right level*.

LLMs evolve rapidly—new architectures, novel attention mechanisms, and hardware-driven execution tricks appear almost monthly. A system like vLLM cannot hardcode its optimizations for one model family or hardware type; instead, it must provide a flexible yet unified framework that supports arbitrary architectures and model sizes, while still delivering system-wide throughput, fairness, and resource efficiency.

Since vLLM's overarching goal is to maximize throughput while preserving fairness across concurrent requests, it needs to handle many users, diverse workloads, and variable model sizes without starving smaller requests or overloading GPU memory. The challenge is that no single layer of the system can address all optimization needs. Some optimizations apply universally across models (such as batching and caching), while others are tied to model architecture (like attention kernels) or even specific hardware instructions (for example, tensor cores).

To resolve this tension, vLLM employs a *layered optimization strategy*, organizing optimization responsibilities across four main levels:

`Scheduler`*: System-wide, model-agnostic optimizations*
: The `Scheduler` is responsible for fairness, efficiency, and scalability at the system level.

`ModelExecutor`: *Model architecture-specific optimizations*
: While the `Scheduler` remains model-agnostic, the `ModelExecutor` understands the details of each model architecture. For example, it applies fused attention kernels for Transformer-based models or specialized operators for multimodal encoders. This level isolates architecture-aware optimizations so they can evolve independently from system-level scheduling.

Model layer: Component-specific optimizations
: At the model architecture's layer level (for example, the attention layer and feedforward block), optimizations are tailored to computational bottlenecks. Techniques such as KV cache reuse, flash attention, and layer-wise operator fusion happen here. This design ensures that optimizations that target a specific subcomponent don't leak into the system-wide scheduling logic.

`CustomOp`: *Hardware-specific optimizations*
: Finally, `CustomOp` captures optimizations for the underlying hardware, such as CUDA kernels, tensor-core acceleration, or quantized operators. By isolating them here, vLLM can take advantage of new GPU capabilities and accelerators without changing its higher-level scheduling or model logic.

This layered strategy avoids cluttering the `Scheduler` with model-specific details while still allowing deep specialization where it matters. It also makes vLLM future-proof: as new architectures or hardware arise, optimizations can be slotted into the right layer without redesigning the entire system.

TensorRT-LLM

TensorRT-LLM is NVIDIA's open source library for high-performance LLM inference on NVIDIA GPUs. From a model checkpoint, it builds highly tuned TensorRT (TRT) engines and provides Python/C++ runtimes with modern serving capabilities, including in-flight batching (TRT-LLM version of continuous batching), paged KV cache, speculative decoding, multi-precision quantization (FP8/FP4/INT4/INT8), and tensor/pipeline parallelism. It integrates tightly with the NVIDIA serving ecosystem—most notably Dynamo and Triton.

Let's look at a quick example with TensorRT-LLM's high-level LLM API:

```
llm = LLM(model="Qwen/Qwen3-7B")

# Sample prompts.
prompts = [
   "Hello, my name is",
   "The capital of France is",
   "The future of AI is",
]
```

```
# Create sampling params.
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)

# Run model generation request
for output in llm.generate(prompts, sampling_params):
   print(
       f"Prompt: {output.prompt!r}, Generated text: {output.outputs[0].text!r}"
   )
```

A core objective of TensorRT-LLM is to demonstrate the maximum practical performance of NVIDIA GPUs for LLMs and to illuminate how to fully exploit Tensor Cores and CUDA kernels.

TensorRT-LLM is best for organizations that have standardized on NVIDIA's hardware and serving tech stack and are seeking best-in-class efficiency and throughput in production.

SGLang

SGLang is a newer entrant, targeting structured-generation and agent applications. SGLang is an open source, high-performance serving framework for LLMs and vision-language models (VLMs). It co-designs a fast backend runtime (kernels, caching, scheduling) with a flexible frontend language and API (which is OpenAI-compatible and native) to make generation faster and more controllable.

SGLang's headline features (available on GitHub) include RadixAttention (for prefix/KV reuse across calls), continuous batching, paged attention/KV, speculative decoding (EAGLE-2/3), chunked prefill, structured outputs (JSON/regex/EBNF), multi-LoRA batching, and parallelism modes (tensor/pipeline/expert/data). A quick example:

```
# load Qwen3 model
llm = sgl.Engine(model_path="Qwen/Qwen3-7B")
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]

sampling_params = {"temperature": 0.8, "top_p": 0.95}

# Run model generation requests
outputs = llm.generate(prompts, sampling_params)
for prompt, output in zip(prompts, outputs):
    print("===============================")
    print(f"Prompt: {prompt}\nGenerated text: {output['text']}")
```

SGLang targets a broader range of hardware than TensorRT-LLM does. Its official documentation and examples cover NVIDIA (H100/Blackwell), AMD Instinct, CPU, TPU, Jetson Orin, and Ascend deployments, making it attractive for multi-vendor fleets.

Functionally, SGLang can be viewed as a peer to vLLM: it emphasizes infrastructure portability, broad open-model support, aggressive performance optimizations (for example, continuous batching, speculative decoding, and prefix/KV reuse), scale-out routing, and strong support for agentic, multistep prompts.

In practice, SGLang's performance is competitive—sometimes better on certain workloads—while vLLM currently enjoys a larger open source community and ecosystem.

Llama.cpp

Llama.cpp is a lean, open source C/C++ inference stack built to run modern open-weight LLMs efficiently on almost any machine—from laptops and workstations to on-prem servers and edge devices. Models are packaged in the GGUF format (typically converted from Hugging Face checkpoints with the repo's scripts). You can serve them via a built-in, OpenAI-compatible HTTP server or drive them from small CLI tools for experiments and benchmarking.

The project emphasizes minimal dependencies, fast startup, aggressive integer quantization (for example, 8/6/5/4-bit), and a portable set of CPU/GPU backends (CPU with SIMD, plus optional accelerators such as Metal, CUDA/ROCm, or Vulkan).

What distinguishes llama.cpp is its "runs anywhere" ethos and tiny operational footprint. Its primary goal is to deliver state-of-the-art local inference with minimal setup and solid performance across a wide range of hardware, including low-cost or low-profile devices, both offline and in the cloud. By contrast, the other three serving frameworks covered here—vLLM, TensorRT-LLM, and SGLang—optimize first for high-throughput, low-latency serving on datacenter GPUs, such as multi-tenant batching, advanced scheduling, and large-scale deployment features.

Llama.cpp prioritizes portability, simplicity, and cost efficiency over absolute peak throughput, making it ideal for local development, privacy-sensitive workloads, edge deployment, and budget-constrained environments. However, it still offers optional GPU acceleration when available.

Let's see a quick example for running the Qwen3 model on a CPU device with llama.cpp:

```
from llama_cpp import Llama
# Run Qwen3 model on CPU with LlamaCPP

# load Qwen model (in GGUF format) from HuggingFace
```

```
llm = Llama.from_pretrained(
   repo_id="Qwen/Qwen3-8B-GGUF",
   filename="*Q8_0.gguf",
   verbose=False
)

# Run LLM generation with high-level API
output = llm(
     "Q: Name the planets in the solar system? A: ", # Prompt
     max_tokens=32, # Generate up to 32 tokens
     stop=["Q:", "\n"], # Stop generating just
          # before the model would generate a new question
     echo=True # Echo the prompt back in the output
)

# Example for chat completion API
output = llm.create_chat_completion(
    messages=[
        {
            "role": "system",
            "content": "You are an assistant who perfectly describes "
                       "images.",
        },
        {
            "role": "user",
            "content": "Describe this image in detail please.",
        },
    ]
)
```

Because llama.cpp brings LLM serving to low-profile devices (CPUs and edge hardware), it sees strong adoption in the three following scenarios:

- Local development, where you can run the built-in OpenAI-compatible server on your machine and reuse existing clients (for example, RAG and search systems)
- Private and on-prem assistants, keeping all data inside your VPC or device boundary
- Edge inference on laptops, desktops, Apple Silicon, and small servers—enabled by aggressive 2- to 8-bit quantization and portable CPU/GPU backends

If you want to expose llama.cpp through a REST API call, you can use Ollama, a higher-level framework that wraps around llama.cpp (and sometimes other backends, like Mistral.cpp or RWKV runners) to make local LLM use simple, consistent, and developer-friendly.

Not every LLM application needs high-throughput serving. When you run inference locally (on-device or on-prem edge), the goals shift:

- You want to optimize for latency and responsiveness, not fleet-wide tokens per second.
- Low concurrency (often a single user) makes large batching and complex schedulers less valuable.
- Footprint and cost (in memory, compute, and power) become primary constraints.
- Privacy and offline reliability are first-class requirements.

Llama.cpp fits this profile: it can run open-weight models directly on CPUs, Apple Silicon, and small GPUs, and exposes an OpenAI-compatible server for drop-in integration. The result is a super-low-cost, low-ops serving option for local development, private and on-prem assistants, and edge deployments—without per-token cloud charges or data leaving the device.

Selecting the Right Framework

The general idea is to choose the framework that best matches your service-level objectives (SLOs), hardware, and operational reality—not the one with the flashiest benchmark. In other words: workload first, not hype.

The evaluation approach we generally recommend is:

1. Start with SLOs, not features. Write down your targets for latency (TTFT, p95/p99), throughput (TPS/QPS), cost per token, quality constraints (structured JSON, safety), and availability.
2. Analyze the real prompts from your use case: make it clear whether it is prefill-heavy or decode-heavy; what's the context length; and whether it includes tool calls and/or multi-turn chains.
3. Make it an apples-to-apples comparison by using the same model, dtype/quantization, max sequence, batch/concurrency, and streaming settings across contenders.
4. Measure operability, using metrics such as cold-start time, observability, autoscaling behavior, multi-tenancy fairness, upgrade friction, and failure modes.
5. Account for hardware and vendor lock-in. If you use multiple vendors (for instance, NVIDIA/AMD/CPU/TPU/edge), portability weighs heavily.
6. Plan for change. Model swaps and new decoding tricks happen weekly, so opt for frameworks you can update without major surgery.

Following our own advice, we'll conclude our library recommendation as follows:

- If you want the fastest path to production, broad model coverage, and a Python-native workflow, choose vLLM. It offers strong baseline performance, continuous batching, and a large community and ecosystem.
- If your workload involves agentic or multistep workflows, strict JSON/regex (grammar-constrained) outputs, and multi-vendor portability, pick SGLang. You get prefix/KV reuse, speculative decoding, and a router for scale-out clusters.
- If you're all-in on NVIDIA and its serving stack and want peak tokens per dollar at scale, start with TensorRT-LLM. First-class Triton and Dynamo integration and deep CUDA/Tensor Core optimizations make it the efficiency leader on NVIDIA hardware.
- If you need local, on-prem, or edge serving with a tiny footprint, low cost, and privacy by default, use llama.cpp. GGUF quantization (2- to 8-bit) and portable CPU/GPU backends make it ideal for laptops, desktops, and small servers.

Keep an Open Mind When Evaluating Serving Frameworks

LLM serving evolves fast—new architectures, kernels, schedulers, quantization schemes, and runtime features land every month. Frameworks must adapt just as quickly to incorporate these innovations.

As serving engineers, we evaluate and reassess different serving frameworks every three to six months and select the option that best fits our current SLOs and hardware reality. We also need to build a framework abstraction layer and have a clear exit plan that lets us swap frameworks without rewriting our apps.

The goal isn't to crown a permanent winner—it's to remain flexible so our system can ride the next wave, not be capsized by it.

Summary

In this chapter, we outlined why general-purpose ML serving stacks fall short for LLMs—chiefly because they lack LLM-specific feature support such as token-level scheduling, KV-cache management, long-context memory handling, and streaming-first execution.

We then took a deep dive into vLLM, covering its architecture, request scheduling and prioritization, and layered optimization strategy, which separates system-wide scheduling from model- and kernel-level specialization. We rounded out the survey with three complementary frameworks: TensorRT-LLM for pushing peak efficiency on NVIDIA GPUs, llama.cpp for lightweight local/on-prem/edge serving,

and SGLang for agentic, multistep workflows and structured outputs across diverse hardware.

The core takeaway we offer you is that your framework choice is contextual. Many teams begin with vLLM for online serving and llama.cpp for local development, but specific requirements—like edge deployment, NVIDIA-centric optimization, strict JSON/grammar outputs, or agent pipelines—may point you to TensorRT-LLM or SGLang instead. As the ecosystem evolves, it's a good idea to stay framework-agnostic. In practice, this means you should do periodic framework research, prefer portable interfaces, and be ready to swap components so your serving stack can adapt to changing business and technical needs.

CHAPTER 9
LLM Optimization in Practice

Optimization is a moving target: in different environments, the "best" strategy changes. Resources are limited, so you can't brute-force every option. To help you optimize efficiently for your own domain, we've carefully chosen some real examples to show how key factors—hardware configuration, model choice, memory and KV-cache behavior, distributed serving and traffic patterns—affect serving performance, and how to measure and interpret those differences. This understanding will give you the intuition to navigate your constraints and converge on a strong serving setup.

In this hands-on chapter, we put everything you learned in the previous chapters to work. Using the open source Qwen3-14B model with vLLM as our running example, we'll walk you through a practical LLM serving optimization process and show how to scale serving both horizontally and vertically.

We'll start with a concise optimization plan and execute it step by step—setting up the environment, preparing the evaluation workload, running experiments, serving the model on both single- and multi-GPU setups, analyzing results, and applying the techniques introduced earlier. We'll conclude with a set of battle-tested takeaways and trade-off recommendations drawn from our own experience, which can serve as guiding principles for your future optimization work.

After reading this chapter, you'll have a clear picture of how LLM serving optimization is done in practice—and the confidence to optimize an LLM for your own use case and traffic patterns.

Lab Code

You can find the lab code (and setup instructions) for this chapter in the *ch09* folder of the book's code repository. Since this lab requires a high-end GPU (such as an NVIDIA L40S) and, in some cases, multi-GPU hardware, it's likely that many readers won't have access to the necessary compute resources.

To make this chapter more accessible, we've prepared a Jupyter notebook that includes not only the complete lab code but also the captured outputs from each optimization step—allowing you to follow the process and results even without running the experiments yourself.

LLM Serving Optimization Plan

In this exercise, we focus on optimizing the *token throughput* of the Qwen/Qwen3-14B model in an online model serving environment. Specifically, our goal is to maximize the serving throughput of a single model instance—processing as many tokens as possible within a given time frame. Higher token throughput directly translates to lower serving costs, since processed tokens form the basis of most LLM pricing models. This optimization goal aligns with a common objective in real-world LLM serving scenarios: to achieve the highest possible token-processing efficiency per unit of time.

Trade-offs Between Throughput and Latency

In many cases, optimization techniques improve throughput and latency simultaneously—for example, by reducing redundant computation or improving batching efficiency. However, for certain traffic patterns, *peak throughput* and *minimal latency* are inherently conflicting goals.

The reason is that higher throughput often relies on batching or queuing requests to better utilize hardware resources, which introduces waiting time and thus increases per-request latency. Conversely, optimizing purely for the lowest latency typically means processing requests as soon as they arrive with reduced parallelism, leading to underutilized resources and lower overall throughput.

In practice, we generally prioritize throughput improvement while keeping latency within an acceptable range, ensuring we gain efficiency without compromising user experience. In this chapter, although our focus is on throughput optimization, we will also discuss techniques (such as distributed serving) that favor low latency when responsiveness is more critical than efficiency.

Here is the high-level optimization plan we will execute in the next section. Whether you execute these steps alongside us or follow along in the Jupyter notebook, this exercise will give you some practical experience with benchmarking and with comparing different optimization techniques and configurations to understand their impact on serving performance:

1. Examine the hardware
: We start by reviewing the hardware configuration and identifying the key GPU specifications that most influence serving performance—such as GPU memory size, bandwidth, compute capability, and NVLink interconnects. Understanding these factors will help you interpret benchmark results more accurately later.

2. Generate benchmark traffic
: Next, we design representative benchmark traffic that simulates real-world request patterns. We'll discuss the types of input data, request length distribution, and repetition settings typically used for caching evaluation.

3. Define evaluation metrics
: Before running the benchmarks, we establish the primary metrics we'll use to measure and compare results—such as token throughput (TPS), time to first token (TTFT), and end-to-end latency. Clear metrics allow for consistent evaluation across different optimization setups.

4. Set up the model serving service
: We then start the model serving service using vLLM and configure it for our Qwen3 model. This includes preparing the environment; verifying GPU availability, memory consumption, and KV cache size; and confirming that the model loads successfully.

5. Benchmark the Qwen3 model with vLLM
: With the default setup in place, we run benchmark traffic against the Qwen3-14B model using the standard vLLM configuration. We collect and analyze the throughput and latency results to establish a baseline.

6. Benchmark the quantized Qwen3 model with vLLM
: Next, we repeat the benchmark using a quantized version of the model—Qwen3-14B-AWQ. By comparing results against the baseline, we demonstrate how model quantization impacts both performance and efficiency, particularly in GPU memory usage and throughput.

7. Apply additional optimization techniques
: After the baseline and quantized experiments, we discuss potential optimization strategies such as chunked prefill, speculative decoding, advanced caching, and batching adjustments. You can follow the benchmarking and analysis process introduced here to measure their effects yourself.

8. Benchmark the Qwen3 model with distributed serving
: Finally, we extend our experiments to distributed-serving setups. Running benchmarks across multiple GPUs and different types of GPUs can help you build your intuition about when distributed serving becomes beneficial and how to best utilize multi-GPU servers for higher scalability.

Optimize Qwen3-14B serving with vLLM

With the plan in mind, let's walk through each step in detail. You can find the complete code and corresponding outputs in the chapter's code repository: *model_optimization_in_practice.ipynb*.

Step 1: Examine the GPU hardware

In this lab, we choose to run our optimization code on an AWS EC2 g6e.2xlarge instance, which has a single NVIDIA L40S GPU. We can use the command-line utility `nvidia-smi` (NVIDIA System Management Interface, which provides real-time monitoring and control of NVIDIA GPU devices) command to inspect the GPU, including:

- Checking available GPUs, memory, and temperature
- Monitoring GPU utilization and running processes
- Identifying driver and CUDA versions
- Diagnosing performance or resource bottlenecks

If you use an NVIDIA GPU, `nvidia-smi` will often be the first command you run before benchmarking or deploying LLM serving workloads to ensure the GPU is recognized, properly configured, and idle before use. Figure 9-1 shows the output from this command.

The key information we normally check is circled in Figure 9-1. It includes:

CUDA and GPU driver version
: We need to make sure they are compatible with our chosen serving framework.

Performance state
: `P8` indicates the GPU is idle. During active serving, you'll see `P0` or `P1`, showing full-speed operation.

Power usage and GPU utilization
: These numbers reflect how well the serving engine saturates the GPU. For example, low utilization under load indicates batching or scheduling inefficiency, while high power but low throughput means there's a memory bottleneck or kernel inefficiency.

Memory usage
: Shows how much GPU memory is currently allocated. `0Mib/46068MiB` (46 GB) reveals the GPU's total memory is around 46 GB and is not currently occupied.

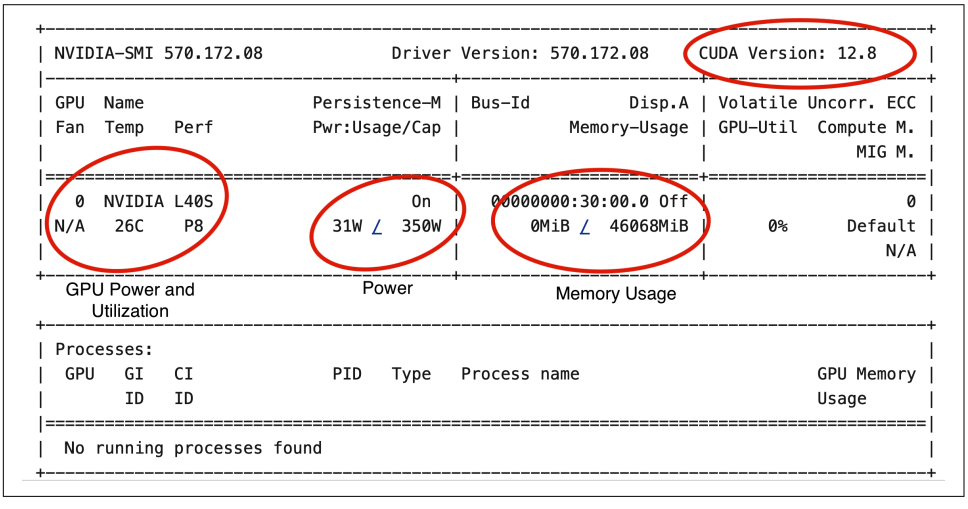

Figure 9-1. Output from the `nvidia-smi` command on an AWS EC2 g6e.2xlarge instance

You can use flags to display only the GPU attributes that you're interested in. For example:

```
nvidia-smi --query-gpu=name,compute_cap,memory.free,memory.used,memory.total \
   --format=csv

>> output:
name, compute_cap, memory.free [MiB], memory.used [MiB], memory.total [MiB]
NVIDIA L40S, 8.9, 45469 MiB, 0 MiB, 46068 MiB
```

Step 2: Generate Benchmark Traffic

Dataset selection is critical to model serving optimization. The entire optimization process aims to tune serving configurations to perform best for specific types of traffic. For example, workloads dominated by long input contexts (which are prefill-heavy) may favor different strategies than those with long outputs (which are decode-heavy). Therefore, the benchmark dataset must accurately reflect the expected usage patterns of the deployed model.

In this case, we want the Qwen3 model to perform well in chatbot and conversational scenarios, where the input and output token lengths are relatively balanced. Users may submit repetitive or follow-up questions. To simulate this usage realistically, we'll use two complementary benchmarking datasets, ShareGPT and Prefix Repetition:

ShareGPT
: The ShareGPT dataset consists of real conversational data collected from users interacting with large language models on platforms such as ShareGPT. It includes a wide variety of user prompts and model-generated responses, covering diverse dialogue structures and linguistic styles. This dataset provides a realistic approximation of end-user traffic, making it ideal for evaluating interactive serving performance and user experience.

Prefix Repetition
: The Prefix Repetition dataset is a synthetically constructed benchmark where each prompt begins with a list of common prefixes, followed by repeated or similar suffixes. This dataset is specifically designed to test a model's *repetition bias*—its tendency to reproduce input text—and to evaluate how well the serving system handles repetitive or copy-prone patterns. It also helps assess cache utilization and decoding robustness.

In this benchmark, we use the ShareGPT dataset to capture natural, diverse conversational patterns and evaluate the model's real-world serving efficiency and responsiveness. We use the Prefix Repetition dataset to probe serving caching behavior under repetitive conditions.

To help you better understand the dataset, we wrote a helper function, `inspect_dataset.py`, to show statistics about a given dataset, including some sample data records:

```
# Use the inspect_dataset.py script to sample 100
# records from the ShareGPT dataset
!python3 inspect_dataset.py \
   --dataset-name sharegpt \
   --dataset-path ShareGPT_V3_unfiltered_cleaned_split.json \
   --model Qwen/Qwen3-14B \
   --num-prompts 100 \
   --save-samples
```

The `inspect_dataset.py` function displays the sample prompt and shows the overall statistics about the prompts, the expected output length, and a histogram view of the prompt-token length distribution. This information will be crucial for understanding the input data we will use in the following serving benchmark.

The following output shows the statistics of 100 samples from the ShareGPT dataset:

```
=== Dataset Overview ===
Total samples: 100

=== Prompt Length Distribution ===
Min prompt length: 5
Max prompt length: 817
Mean prompt length: 232.60
Median prompt length: 141.50
Std prompt length: 241.42
```

```
=== Output Length Distribution ===
Min output length: 4
Max output length: 771
Mean output length: 220.61
Median output length: 164.50
Std output length: 210.23

=== Sample Prompts ===
--- Sample 1 ---
Prompt length: 24
Output length: 770
Request ID: 1
Prompt: help me create a rust app that supports the
elevenlabs.io api and that can read the contents of
clipboard aloud using tts
...

=== Prompt Length Histogram ===
   5-  95 tokens: **********************************************
  95- 185 tokens: ********
 185- 275 tokens: *********
 275- 365 tokens: **************
 365- 456 tokens: *****
 456- 546 tokens: *****
 546- 636 tokens: ****
 636- 726 tokens: ***
 726- 817 tokens: *******

Saved 100 samples to sharegpt_samples.json
```

You can use the same function to examine the Prefix Repetition dataset. Since this dataset is synthetic and designed for evaluating cache performance, we can precisely control parameters such as the prefix length, suffix length, and the number of unique prefixes. Using fewer prefixes results in more repetition across input prompts (in the dataset), creating a stronger cache-reuse traffic pattern:

```
python inspect_dataset.py \
   --dataset-name prefix_repetition \
   --model Qwen/Qwen3-14B \
   --num-prompts 50 \
   --prefix-repetition-prefix-len 256 \
   --prefix-repetition-suffix-len 256 \
   --prefix-repetition-num-prefixes 5 \
   --prefix-repetition-output-len 128 \
   --save-samples
```

The Prefix Repetition dataset creates synthetic requests where each request has a repeated prefix pattern (for example, the prefix "The quick brown fox" might be repeated multiple times). The prefix is followed by a unique suffix to make a prompt.

From a test traffic perspective, we use the vLLM benchmark CLI tool (bench serve) to generate controlled benchmarking traffic. This tool allows us to configure parameters such as the total number of prompts, request rate, traffic ramp-up patterns, and maximum number of concurrent requests. It also automatically collects key performance metrics, including latency and throughput, which are essential for evaluating serving performance.

In the following example, we sample 2,000 prompts from the ShareGPT dataset and send them to a local vLLM server at a rate of 10 requests per second, simulating realistic conversational traffic under moderate load:

```
!vllm bench serve \
     --backend vllm \
     --base-url "http://localhost:8000" \
     --dataset-name sharegpt \
     --dataset-path ShareGPT_V3_unfiltered_cleaned_split.json \
     --num-prompts 2000 \
     --request-rate 10 \
     --burstiness 1.0 \
     --save-result \
     --append-result \
     --result-filename test_serve_results.txt \
     --model Qwen/Qwen3-14B \
     --max-concurrency 10
```

At this stage, we have established the methodology for generating and sending benchmark traffic to assess real-world serving efficiency and responsiveness, and for probing caching behavior under repetitive workloads.

Step 3: Define Evaluation Metrics

Table 9-1 lists commonly used metrics for LLM serving performance.

Table 9-1. Primary metrics used in LLM serving performance evaluation

Category	Primary metrics
Throughput	Total token throughput (TPS), output token throughput (TPS), request throughput (number/s)
Latency	TTFT, TPOT, ITL (mean + P99)
Resource utilization	GPU utilization, memory usage
Workload profile	Input/output token ratio, concurrency, request rate
Reliability / cost	Error rate, cost efficiency

To keep things simple in this lab, we will focus on the following four metrics to measure throughput and latency performance in our benchmark tests:

Throughput metrics

Total token throughput (TPS)
: The combined rate of input and output tokens processed per second. This is a high-level indicator of system efficiency.

Output token throughput (TPS)
: The average number of output tokens generated per second. This is a core metric for measuring *decoding performance*—the main driver of LLM cost.

Latency metrics

Mean TTFT (ms)
: The average time to generate the first token after a request starts. This metric captures *prefill efficiency*—which is influenced by model loading, tokenization, and scheduling delay.

Mean ITL (ms)
: The average delay between consecutive output tokens streamed to the client. This affects the user experience in real-time streaming quality, which is critical for chat and interactive agents.

Step 4: Set Up the Model Serving Server

The last thing before we head into the actual optimization experiments is to start the vLLM server and host the LLM. To create a baseline, we launch a vLLM server with its default settings to host the Qwen3/Qwen3-14B model:

```
vllm serve Qwen/Qwen3-14B
//Or
proc = start_vllm_serve(model="Qwen/Qwen3-14B")
```

During model loading, the vLLM server logs its GPU memory consumption, which you can find in the *vllm.log* file. Here is an example:

```
Loading weights took 4.47 seconds
Model loading took 27.5185 GiB and 5.265852 seconds
Available KV cache memory: 11.00 GiB
GPU KV cache size: 72,064 tokens
Maximum concurrency for 40,960 tokens per request: 1.76x
```

From the logs, you could see that vLLM occupies 38.5GiB out of 46 GB of GPU memory to host the Qwen3/Qwen3-14B model. Among that, the model takes up 27.5185 GB and the KV cache takes up 11 GB for 72,064 tokens.

This breakdown shows that the model itself occupies more than 65% of total GPU memory, leaving relatively little space for KV caching. In LLM serving, KV cache capacity directly limits batching and concurrency. When cache memory is constrained, the server must either reduce batch size or evict cache entries more

frequently, which leads to more recomputation during decoding. As a result, GPU utilization becomes less efficient, and token throughput decreases.

To improve throughput, we can switch to a quantized model, which significantly reduces the model weight footprint. The freed-up GPU memory can then be allocated to a larger KV cache, enabling higher batch concurrency and more efficient GPU scheduling, ultimately increasing overall serving throughput.

Step 5: Benchmark the Qwen3 Model with vLLM

Let's start sending our test traffic to the vLLM server to benchmark its performance. First, we'll send 2,000 prompts (from the ShareGPT dataset) at a rate of 10 requests per second:

```
!vllm bench serve \
     --backend vllm \
     --base-url "http://localhost:8000" \
     --dataset-name sharegpt \
     --dataset-path ShareGPT_V3_unfiltered_cleaned_split.json \
     --num-prompts 2000 \
     --request-rate 10 \
     --burstiness 1.0 \
     --save-result \
     --append-result \
     --result-filename test_serve_results.txt \
     --model Qwen/Qwen3-14B \
     --max-concurrency 10
```

Once the requests are processed, the overall results (metrics) are summarized as follows. The total throughput for this test run is 474 tokens per second, which is a pretty good number:

```
============ Serving Benchmark Result ============
Successful requests:                     2000
Maximum request concurrency:             10
Request rate configured (RPS):           10.00
Benchmark duration (s):                  1810.09
Total input tokens:                      446619
Total generated tokens:                  412052
Request throughput (req/s):              1.10
Output token throughput (tok/s):         227.64
Peak output token throughput (tok/s):    240.00
Peak concurrent requests:                15.00
Total Token throughput (tok/s):          474.38
---------------Time to First Token----------------
Mean TTFT (ms):                          104.15
...
Mean ITL (ms):                           43.24
Median ITL (ms):                         42.43
P99 ITL (ms):                            72.15
```

Once the conversation-prompt traffic finishes, let's send our prefix-repetition prompts to benchmark the serving's cache performance:

```
!vllm bench serve \
   --backend vllm \
   --base-url "http://localhost:8000" \
   --model Qwen/Qwen3-14B \
   --dataset-name prefix_repetition \
   --num-prompts 1000 \
   --request-rate 5 \
   --prefix-repetition-prefix-len 256 \
   --prefix-repetition-suffix-len 256 \
   --prefix-repetition-num-prefixes 10 \
   --prefix-repetition-output-len 128 \
   --max-concurrency 10 \
   --save-result \
   --append-result \
   --result-filename test_serve_results.txt
```

If you run `nvidia-smi` during the performance run, you will find GPU utilization at 97%, which indicates the GPU is highly saturated. The results of this performance test with Prefix Repetition dataset traffic are:

```
============ Serving Benchmark Result ============
Successful requests:                     1000
Maximum request concurrency:             10
Request rate configured (RPS):           5.00
Benchmark duration (s):                  569.00
Total input tokens:                      512000
Total generated tokens:                  127066
Request throughput (req/s):              1.76
Output token throughput (tok/s):         223.31
Peak output token throughput (tok/s):    240.00
Peak concurrent requests:                17.00
Total Token throughput (tok/s):          1123.13
---------------Time to First Token----------------
Mean TTFT (ms):                          104.64
...
Mean ITL (ms):                           43.95
Median ITL (ms):                         42.82
P99 ITL (ms):                            59.22
```

Compared to the ShareGPT dataset benchmark, the Prefix Repetition dataset has significantly higher throughput—1,123 TPS versus 474 TPS—while maintaining similar average TTFT and ITL values (104.15 ms/43.24 ms versus 104.64 ms/43.95 ms).

This throughput gain occurs under repetitive traffic because vLLM automatically applies several caching and batching optimizations—including prefix caching, continuous batching, and memory block sharing—that reuse computation across similar input prompts efficiently.

Step 6: Benchmark the Quantized Qwen3 Model with vLLM

Although the standard vLLM server already delivers strong performance, we can further improve throughput by freeing up more GPU memory for KV caching. One effective approach is model quantization.

In this section, we'll rerun the same benchmark using the Activation-Aware Weight Quantization (AWQ) version of the Qwen3 model—Qwen/Qwen3-14B-AWQ (4-bit)—and compare its performance metrics against the original Qwen3-14B model to evaluate the impact of quantization on memory usage and serving efficiency.

First, stop the previous vLLM server and then start it again, this time with the Qwen3-14B-AWQ model:

```
!pkill -f "vllm serve"
proc = start_vllm_serve(model="Qwen/Qwen3-14B-AWQ")
```

If you check the server log (*vllm.log*), you'll see that the AWQ 4-bit quantized model occupies only 9.6 GB of GPU memory, compared to the 27.5 GB used by the original Qwen3-14B model. This frees up 17 GB of memory for caching:

```
Loading weights took 4.47 seconds
Model loading took 27.5185 GiB and 5.265852 seconds
Available KV cache memory: 11.00 GiB
GPU KV cache size: 72,064 tokens
Maximum concurrency for 40,960 tokens per request: 1.76x

Model loading took 9.3619 GiB and 10.652314 seconds
Available KV cache memory: 29.15 GiB
GPU KV cache size: 191,056 tokens
Maximum concurrency for 40,960 tokens per request: 4.66x
```

The additional KV cache memory allows vLLM to store more than twice as many tokens—increasing from 72,064 to 191,056 tokens. This can significantly improve throughput by reducing prefix recomputation and enabling larger batch processing.

After running the ShareGPT traffic, we obtain the benchmark results shown in Figure 9-2. Under the ShareGPT traffic, the quantized model delivers a substantial performance boost over the original Qwen3-14B model. The total throughput increases by 2.7x, from 474 TPS to 1,280 TPS, while the average TTFT improves by nearly 42%, decreasing from 103.61 ms to 59.29 ms.

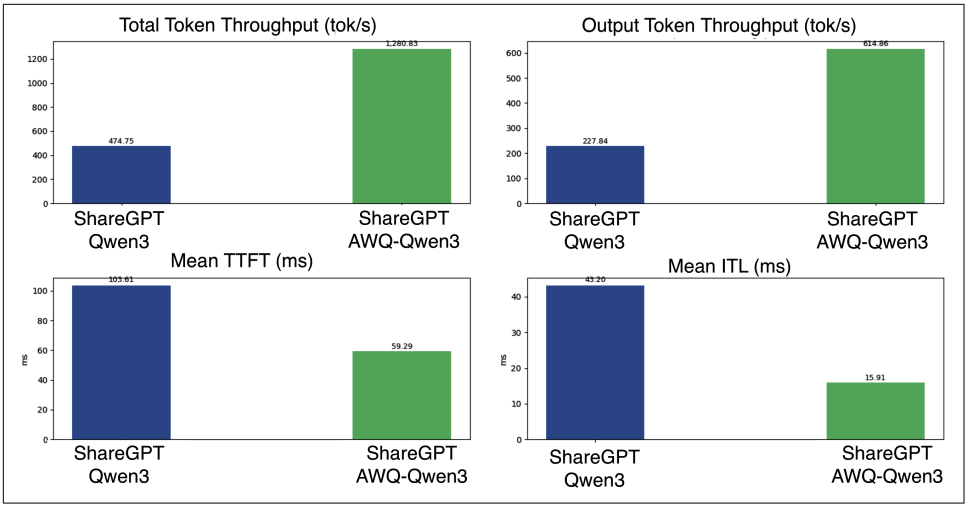

Figure 9-2. Throughput and latency comparison between the Qwen3-14B model and its AWQ 4-bit quantized version, evaluated on the ShareGPT dataset

A similar improvement is observed on the Prefix Repetition dataset, where the quantized model consistently delivers higher throughput and lower latency, as shown in Figure 9-3.

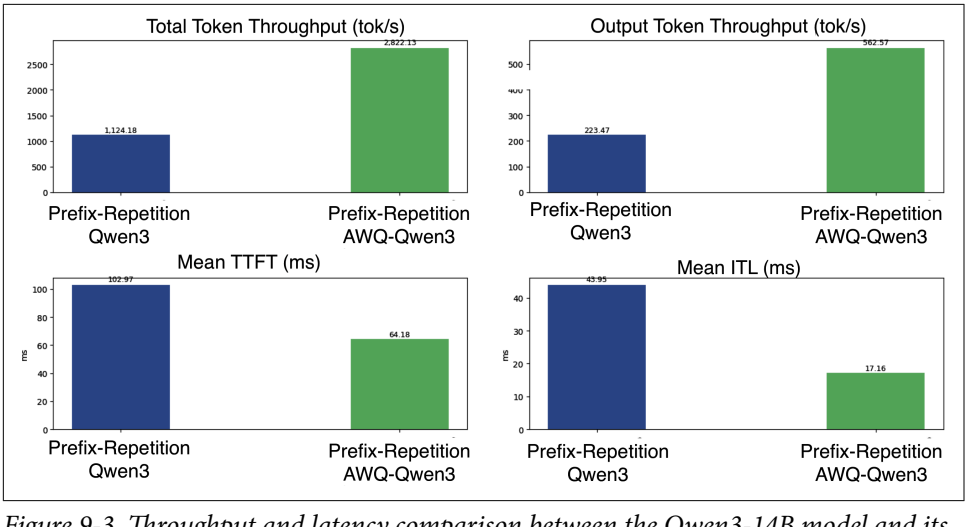

Figure 9-3. Throughput and latency comparison between the Qwen3-14B model and its AWQ 4-bit quantized version, evaluated on the Prefix Repetition dataset

The AWQ quantization results in this experiment primarily demonstrate how weight quantization reduces data movement between CPU and GPU, as well as overall memory usage. For techniques that also reduce compute—specifically activation quantization—please refer to Chapter 6.

Step 7: Apply Additional Optimization Techniques

Although hosting a quantized model on vLLM with default configurations offers strong performance in general, we could push the limits further by applying traffic-pattern-specific or GPU-specific optimization techniques and by fine-tuning the vLLM configurations for these techniques.

At this stage, the model serving service is normally already performing well. Now we need to optimize its serving performance for a specific type of workload or GPU. For example:

- For long-context, prefill-heavy workloads, most computation happens during input processing. We could try enabling the LMCache, which allows us to reuse previously computed KV pairs for repeated prefixes, greatly improving efficiency for prompts with shared context (for example, multi-turn chat or RAG).
- For decode-heavy workloads, where outputs are long, speculative decoding is more effective. It speeds up generation by using a smaller draft model to predict multiple tokens ahead, reducing decoding iterations and boosting throughput.

Please check out the hands-on example in Chapter 7 for more on how to use LMCache and speculative decoding in practice.

Once we choose the optimization techniques for our workload, we can work on figuring out their optimal configurations. Luckily, most of the model serving framework exposes lots of knobs, so you can easily take fine control over its serving behavior. For example, in vLLM, you can adjust the KV cache size and block management for better caching with the following configuration:

```
# Use more GPU memory for KV cache
--gpu-memory-utilization 0.9
# Increase context length if needed
--max-model-len 4096
# Smaller blocks = better cache utilization
--block-size 16
```

The following configuration allows fine-grained control over batching:

```
# More concurrent requests
--max-num-seqs 512
# Larger token batches
--max-num-batched-tokens 16384
# Allow more padding for batching
--max-paddings 256
```

A typical vLLM server startup script, after applying fine-tuned configuration parameters, looks like this:

```
proc = start_vllm_serve(
   model="Qwen/Qwen3-14B-AWQ",
   extra_args=(
        "--quantization awq "
        "--gpu-memory-utilization 0.95 "
        "--max-model-len 1024 "
        "--block-size 16 "
        "--enable-prefix-caching "
        "--max-num-seqs 8 "
        "--max-num-batched-tokens 8192 "
        "--enable-chunked-prefill "
   )
)
```

Now that you've seen how vLLM configurations can be tuned to better fit your specific use case, you can try the following exercise if you have time.

Exercise: Fine-Tune Your vLLM Serving

If you'd like to explore further, try this hands-on exercise. Using the same benchmark traffic and evaluation methods introduced in this chapter, experiment with different vLLM configurations to fine-tune your serving performance.

Adjust parameters such as batch size, cache size, and scheduling options to observe their impact on throughput and latency. Record and compare your results to understand how each setting influences overall efficiency.

Don't Overtune Your Configuration

While tuning vLLM configuration can yield impressive performance gains, avoid overoptimizing for a single setup. A configuration that's perfectly tuned for one GPU type or workload pattern may perform poorly—or even fail—on different hardware or with varied traffic. Overfitting your serving parameters makes the system less portable and harder to maintain.

Modern model serving frameworks are increasingly intelligent—they can automatically infer good settings at startup based on the hardware and environment. In practice, we spend most of our effort identifying which optimization techniques to apply rather than chasing the perfect configuration.

Step 8: Benchmark the Qwen3 Model with Distributed Serving

In this section, we extend our optimization experiments to a distributed-serving setup. We'll benchmark different distributed configurations and hardware setups to illustrate how distributed serving works in practice and to discuss key trade-offs.

Our examples focus on multi-GPU distributed serving within a single server node, as this represents the most common and practical (non-hyperscale) LLM-serving scenario. For multi-node LLM setups, we recommend trying prefill-decode disaggregation.

To demonstrate the performance impact of GPU architecture differences, we will run the 4-bit quantized Qwen3-14B-AWQ model on two different EC2 instances: g6e.12xlarge and p4d.24xlarge. The g6e.12xlarge instance is equipped with four NVIDIA L40S GPUs, while the p4d.24xlarge instance includes eight NVIDIA A100 GPUs.

Although the L40S generally outperforms the A100 in single-GPU inference tasks, our distributed serving experiments on the p4d instance show lower latency than those on the g6e. We'll explore the reasons behind this difference in the following discussion.

Host model in a distributed setup

Running a model across multiple GPUs with vLLM is straightforward. The `vllm serve` command shows how to host the Qwen3-14B-AWQ model on two GPUs using tensor parallelism:

```
vllm serve Qwen/Qwen3-14B-AWQ --tensor-parallel-size 2 \
         > vllm.log 2>&1 &
```

With `--tensor-parallel-size 2`, vLLM automatically initializes a distributed-serving group across two GPUs, handling communication and model partitioning internally.

Next, we use the `vllm bench serve` CLI to send benchmark traffic using the ShareGPT and Prefix Repetition datasets, to measure serving performance under realistic workloads.

Next, we'll repeat these distributed-serving benchmarks on both 2-GPU and 4-GPU configurations across the g6e.12xlarge (L40S) and p4d.24xlarge (A100) instances to compare their scaling efficiency and latency behavior.

Distributed serving performance analysis

Let's first examine the benchmark results of the quantized Qwen3-14B model on a g6e.12xlarge server using one, two, and four GPUs, shown in Figure 9-4.

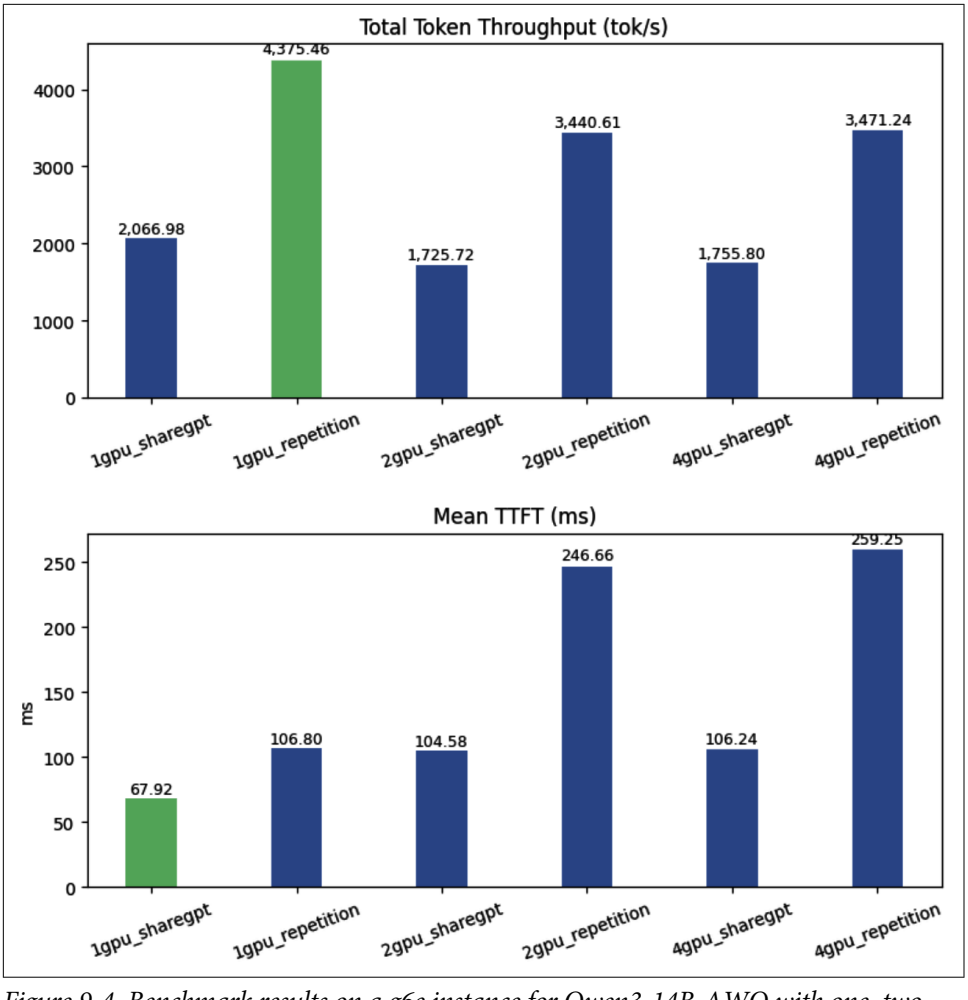

Figure 9-4. Benchmark results on a g6e instance for Qwen3-14B-AWQ with one, two, and four GPUs

You can see that, on the g6e instance, the single-GPU setup delivers the best performance. It achieves both the highest throughput and the lowest TTFT (latency), outperforming the two-GPU and four-GPU configurations.

Next, let's examine the benchmark results on the p4d instance, shown in Figure 9-5.

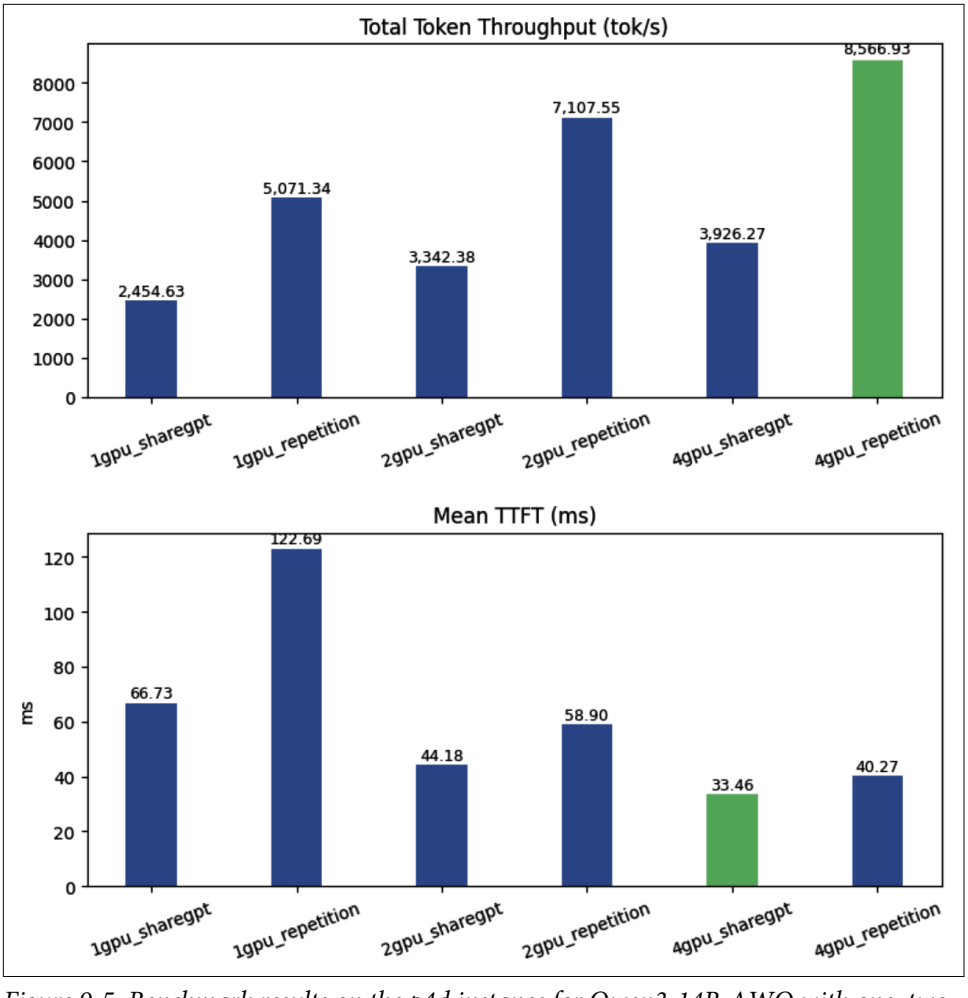

Figure 9-5. Benchmark results on the p4d instance for Qwen3-14B-AWQ with one, two, and four GPUs

In contrast to the g6e results, the p4d instance exhibits the opposite behavior: multi-GPU configurations perform better than a single GPU, with the four-GPU setup achieving the best overall throughput and lowest latency.

Why does multi-GPU serving perform better on p4d but not on g6e, even though the L40S GPUs in g6e are generally more efficient for inference?

The answer lies in the GPU interconnect architecture discussed in Chapter 5. The A100 GPUs in the p4d instance are connected via NVLink, which enables high-speed, low-latency communication between GPUs. This allows tensor parallelism to efficiently split and synchronize model computation across multiple devices.

In contrast, the L40S GPUs in the g6e instance lack NVLink connections and communicate only over PCIe, which has significantly lower bandwidth and higher latency. As a result, multi-GPU setups on g6e suffer from inter-GPU communication overhead, making the single-GPU configuration faster overall.

Distributed Serving Doesn't Always Guarantee Better Performance

A common misconception is that adopting distributed serving will always *improve* performance. However, as Figure 9-4 illustrates, a single GPU can sometimes outperform multi-GPU setups in both throughput and latency.

Even in the p4d benchmark (Figure 9-5), where the four-GPU configuration delivers the best overall throughput and lowest latency, the advantage is not absolute. Running four independent model instances—one per GPU—can achieve nearly triple the total throughput (9,816 TPS versus 3,926 TPS) compared to using four GPUs jointly for a single distributed model.

The true benefits of distributed serving lie in *vertical scaling*—reducing latency and overcoming model size limitations. If your workload demands faster response times or your model (even after quantization) cannot fit into a single GPU's memory, distributed serving becomes essential.

Figure 9-5 shows that the TTFT latency on the p4d instance drops by almost 50%, from 66 ms to 33 ms, when moving from one to four GPUs. Such latency improvements cannot be achieved by horizontal scaling alone—adding more independent model replicas does not reduce per-request latency.

Common Optimization Trade-Offs

In practice, we spend far more effort deciding which optimization techniques to apply than chasing the "perfect" configuration. Understanding the trade-offs—and adapting them to your scenario—is the essence of real-world LLM serving optimization. Here are the five trade-offs we often consider during LLM serving optimization:

Throughput versus latency
: Higher throughput usually comes from batching and scheduling optimizations that better utilize the GPU. However, larger batches introduce waiting time, increasing per-request latency. Interactive applications (like chatbots) favor low latency, while offline or batch inference favors high throughput.

Memory efficiency versus model quality
: Techniques like quantization or weight compression reduce GPU memory usage and expand KV cache capacity. They may, however, introduce small accuracy

losses or token instability. The right balance (e.g., 4-bit versus 8-bit quantization) depends on your tolerance for quality degradation versus performance gain.

Hardware utilization versus flexibility
: Aggressive tuning (batch size, tensor parallelism, cache size) can maximize performance for a specific GPU or workload. But these settings often don't generalize to other hardware or traffic patterns. A more flexible setup trades a few percentage points of efficiency for portability and robustness.

Vertical versus horizontal scaling
: Vertical scaling (distributed serving) enables larger models and can reduce latency by spreading inference across GPUs. Horizontal scaling (multiple single-GPU replicas) improves total throughput and fault tolerance. In practice, most production systems focus on horizontal scaling.

Static optimization versus adaptive serving
: Static configurations are predictable but limited—they can overfit to a specific GPU or workload. Adaptive runtime optimization dynamically adjusts batch size, cache policy, or scheduling based on live traffic and hardware metrics. This represents the next stage of serving frameworks: self-optimizing systems that balance performance automatically.

Summary

In this chapter, we explored how to optimize LLM serving performance using vLLM and the Qwen3-14B model as a practical example. We hope this has helped you build your intuition and methodology for improving throughput, latency, and scalability in your real-world serving systems. Some key takeaways are:

- Optimization is not about finding a universal "best" configuration. It's about understanding your system's dominant bottleneck and applying the right techniques at the appropriate level—hardware, model, or scheduling.
- Start the optimization process by understanding your scenario. Clearly define your business use case, traffic pattern, and performance target before tuning.
- Craft representative test datasets and meaningful metrics that reflect your real user workloads and performance goals—they are critical for meaningful optimization results.
- Begin with general throughput optimizations. Establish a strong baseline using broad efficiency techniques before applying workload-specific tuning.
- Apply targeted techniques for specific traffic types. For example, you could use LMCache for long-context (prefill-heavy) workloads and speculative decoding for long-output (decode-heavy) workloads.

- Balance throughput and latency based on your priorities. You can trade throughput for lower latency by allocating more memory per request or limiting concurrency, ensuring faster response at the cost of total capacity.
- Use distributed serving selectively. It's especially valuable when model size exceeds single-GPU limits or when low latency is the priority.
- For most cases, running on single GPUs and scaling horizontally with multiple replicas offers better throughput and simplicity.
- Avoid overspecific static configurations. Settings like fixed KV cache block sizes or hardware-tied parameters may overfit one GPU type. Modern serving frameworks are increasingly adaptive, automatically tuning batch size, cache policy, and scheduling based on live traffic and hardware metrics.

Optimization Is Iterative

Effective serving optimization comes from continuous experimentation, measurement, and trade-off analysis rooted in real user scenarios.

CHAPTER 10
Advancements in LLM Serving

If you've made it this far, congratulations on going through the journey from understanding model serving paradigms all the way to how to serve LLMs efficiently for different use cases.

The final chapter highlights a few emerging advancements in LLM serving and serves as a guide for your continued learning as new ideas and techniques rapidly evolve. Some of these ideas could fill entire books, and progress in the field is moving quickly. It is an exciting time to witness and contribute to the evolution of intelligent inference systems. Our goal here is to introduce the main ideas and frameworks so that you leave this book feeling equipped to connect the core foundations we have covered with the new ideas shaping the next generation of LLM serving systems.

In this chapter, we will explore:

- Semantic caching and routing as high-level mechanisms for smarter semantic-aware request distribution
- Performance profiling for fine-grained performance tuning
- Multimodal serving, as text-based LLMs expand into vision language models (VLMs) and other modalities
- Edge serving, bringing low-latency, privacy-preserving inference to devices
- Multi-LoRA, enabling scalable, efficient deployment of personalized, fine-tuned models
- LLM serving systems as the backbone of reinforcement learning inference

Semantic Caching

In Chapter 7, we discussed data parallelism where, behind a model serving endpoint, there are multiple model replicas that serve external traffic, with a routing layer in front that performs the load balancing. The need for routing becomes more crucial as we do prefix caching and KV-cache-utilization-aware load balancing.

More and more, serving systems are becoming aware of semantics and are operating at a higher level of the whole ecosystem. Semantic-aware routing, caching, and retrieval are no longer based on exact prompts and a set of model replicas, but instead use embedding and vector search to recognize prompts with the same intent. With that, this semantic routing layer on top enables even more cache hits, smarter decisions about when to enable model reasoning, agentic tool filtering, and model selection at the model-endpoint level rather than purely at the model-replica level, as shown in Figure 10-1.

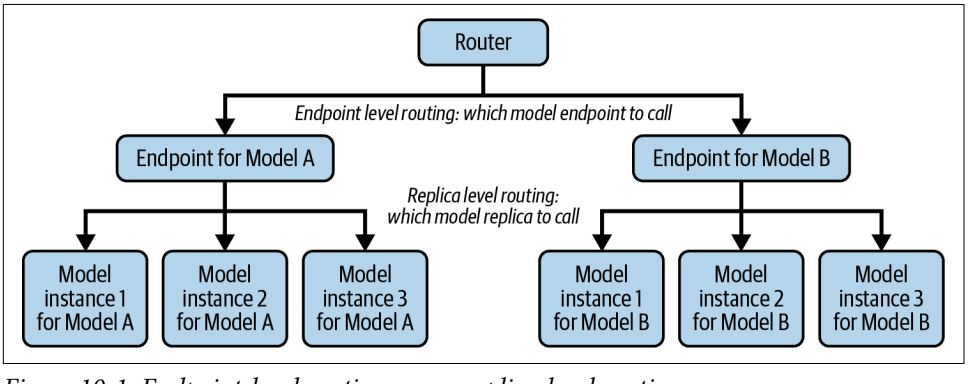

Figure 10-1. Endpoint-level routing versus replica-level routing

The first and simplest reason for using semantic routing is to avoid calling an LLM when the prompts are similar. "How long is the flight from Seattle to Hawaii?" can also be asked as "Tell me how many hours it takes to fly from Seattle to Hawaii." In this case, calling an LLM again, especially an external LLM, is overkill: a waste of money that incurs unneeded latency. Semantic caching can help here by understanding that these queries are similar and storing the prior query results to avoid an extra call to the LLM.

Secondly, even for a new prompt that does not have a cached response, not all questions require a big model or turning reasoning on. A semantic router can call a lightweight encoder model, such as ModernBERT, to decide quickly if a prompt is complex enough to warrant a call to the most advanced model and/or a call with reasoning enabled.

More and more, companies are realizing that task-specific, fine-tuned small language models (SLMs) with around 8 billion to 32 billion parameters can perform on par with or even better than giant general-purpose LLMs, with better reliability and lower latency for SLA/SLO control. A semantic router here can understand the query and route the request to the appropriate model endpoint.

Lastly, in agentic settings, routing service can be tasked with more functionality beyond forwarding requests to the model endpoint. For example, the routing service can perform first-level tool filtering to avoid exposing the model to all available tools. This is increasingly done through Model Context Protocol (MCP), where tools are registered with structured schemas, permissions, and metadata. The router, based on the user requests, selects only the relevant tools and passes a small subset of available tools to the model, which lowers prompt overhead, improves latency, and reduces the risk of unintended tool use. The routing service also serves as a central enforcement point for security, policy, and compliance checks.

Figure 10-2 shows a simple example of a semantic router that accepts requests and provides multiple functionalities to decide which LLM needs to be called.

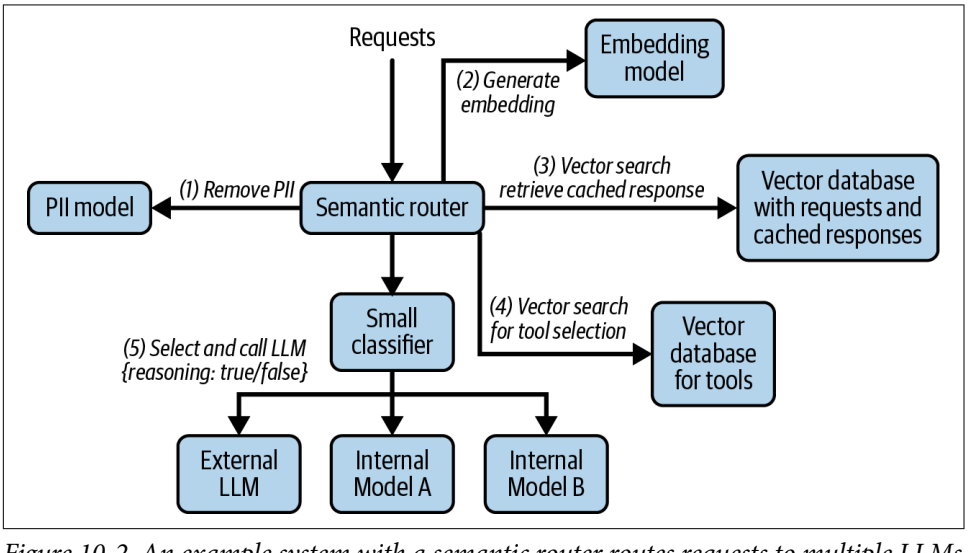

Figure 10-2. An example system with a semantic router routes requests to multiple LLMs

Let's break it down and understand what each numbered arrow is doing:

1. Usually the first step is masking any personally identifiable information (PII). Raw user inputs often contain sensitive data, such as a person's name, address, contact information, credit card number, or health details. Removing or obfuscating that PII is crucial to ensure that downstream logs, caches, and routing decisions never expose private information. It can happen in the router or the

gateway before that. The PII model is usually a small encoder model fine-tuned for named entity recognition (NER) on privacy-sensitive entities. Example output from a prompt can look like this:

```
[
    {"start": 5, "end": 13, "label": "PERSON",
     "text": "John Doe", "confidence": 0.97},
    {"start": 66, "end": 81, "label": "EMAIL",
     "text": "john@abc.com", "confidence": 0.95},
]
```

With the positional information and category, we can go back to our prompt and replace the PII tokens with `<NAME_1>` and `<Email_1>`. In many cases, `regex` methods are also used to complement the NER model in PII masking.

2. The second step shown in Figure 10-2 is the main semantic understanding step. Here, we call an embedding model to transform each prompt into a list of numbers with a length of, say, 768. This length is known as an *embedding vector*. Prompts with similar meanings end up having similar embedding vectors. For example, "forget my login" and "reset password" would have similar embedding vectors, while "forget my login" and "create me a draft article" would have very different embedding vectors.
3. With the embedding vector, we can now perform a vector search to find prior stored prompt and response pairs. If a very close match is found, the cached response is immediately returned without any extra LLM calls.
4. Many agentic use cases supplement LLMs with tools. You can use a vector search to narrow down the list if there are a lot of tools, since it can be expensive to send all tools to the LLM.
5. The embedding vector can be used as an input to a small classifier that decides which LLM to call and whether to enable reasoning. Based on the output label this classifier chooses, the appropriate LLM is invoked. For example, it might call an external state-of-the-art model for complex reasoning, a medium-sized internal model for specific tasks, or a small internal model to answer a simple question.

Performance Profiling Strategies

Because LLM serving is expensive, the art of workload profiling has deepened. This was far less critical with the last generation of predictive models, which had smaller inference workloads. Today, however, as LLMs are deployed across more products and organizations, serving at scale means that a single percentage point of improvement can save millions of dollars in infrastructure costs. Those last few percentage points of serving performance are no longer optional luxuries but operational necessities.

In this section, we explore how to profile your LLM inference workloads to identify performance hotspots and show how low-level kernel optimization can improve throughput and latency. While this book does not dive into writing custom CUDA or Triton kernels, it is essential to understand what they optimize and how profiling connects system-level observations to kernel-level behavior.

Profiling has three major layers: the *serving* layer, *framework* layer, and *runtime* layer:

Serving layer
: You already have a good understanding of the topmost layer—the serving layer. In this layer, you'll obtain metrics such as throughput, TTFT, ITL, GPU utilization, and memory utilization, and make adjustments to improve serving performance and achieve SLA/SLO targets.

Framework layer
: At the framework level, inside deep-learning frameworks such as PyTorch, the model is represented as a graph of operations. Here, tools like PyTorch Profiler let you inspect how specific operators (that is, building-block computations like `matmul`, `attention`, or `layernorm`) execute, how data moves between the CPU and GPU, and whether the pipeline efficiently overlaps computation and input/output.

  For example, the operator execution timeline can tell which specific operator is dominating GPU execution time. You can then target that specific operator and replace it with different kernel implementations. Another example is identifying when the GPU is mostly idling, waiting for the CPU side to finish its work. In this case, you could try to improve GPU utilization by moving as many CPU workloads as possible to async processes and off the critical path.

Runtime layer
: Often, framework-level profiling provides a clear picture of how a model is spending its time and which operators dominate end-to-end latency. But when the slowest operators are dominated by GPU kernel execution, you need to move deeper into runtime-level profiling. A model is ultimately executed as a sequence of operators (such as `matmul`, `attention`, or `layernorm`), each of which expands into one or more CUDA kernels that perform the actual math. Understanding what happens inside these kernels is crucial, because inefficiencies at this level often explain why an operator appears expensive at the framework level.

  NVIDIA Nsight Systems is a common starting point that provides a system-wide performance-analysis timeline to identify major bottlenecks; NVIDIA Nsight Compute, by contrast, provides a microarchitectural view of the kernel-level details.

For example, Nsight Systems can reveal that even when GPU utilization appears high, the kernels are actually separated by large idle gaps. This could be because the host is launching them too slowly or because data transfers are not overlapping with computation. In this situation, the GPU is basically waiting between kernels. To fix this, you could try to keep the GPU busier by overlapping data copies with computation, such as by using more than one CUDA stream.

When a single kernel is the bottleneck and dominates the overall time, Nsight Compute can help explain why. Common reasons include low thread occupancy or frequent memory stalls. For example, perhaps the attention softmax kernel turns out to be the slowest step because it repeatedly loads large intermediate tensors from global memory instead of keeping them in shared memory. With this information, you could target that kernel directly by optimizing its memory access patterns, adjusting its block/grid configuration, or replacing it with a more efficient fused kernel. This book won't go into low-level CUDA kernel optimization, but Nsight Compute gives you the insights you need to explore that further.

In practice, the real challenge of performance profiling is knowing when to use each tool and how to move from one layer to the next. To avoid getting lost in the large ecosystem of profilers, it helps to adopt a clear strategy that guides you from the initial symptoms to the root cause. The decision flowchart shown in Figure 10-3 summarizes this process and provides a practical roadmap for identifying bottlenecks.

The process typically begins with Nsight Systems, which provides a top-down, system-wide timeline. This view helps determine whether the GPU is actually the bottleneck or whether the real issue lies in CPU overhead, data loading, or I/O. If the GPU is underutilized, framework-level profiling with the PyTorch Profiler (in CPU Mode) can reveal where time is being spent on the host side—such as slow preprocessing, Python dispatch overhead, or serialization work.

If the GPU is busy but the overall latency is still unexpectedly high, the next step is to determine which operator is responsible. Running the PyTorch Profiler in CUDA Mode attributes GPU time back to high-level PyTorch operators, allowing you to pinpoint whether `attention`, `matmul`, `layernorm`, or another operator is dominating execution.

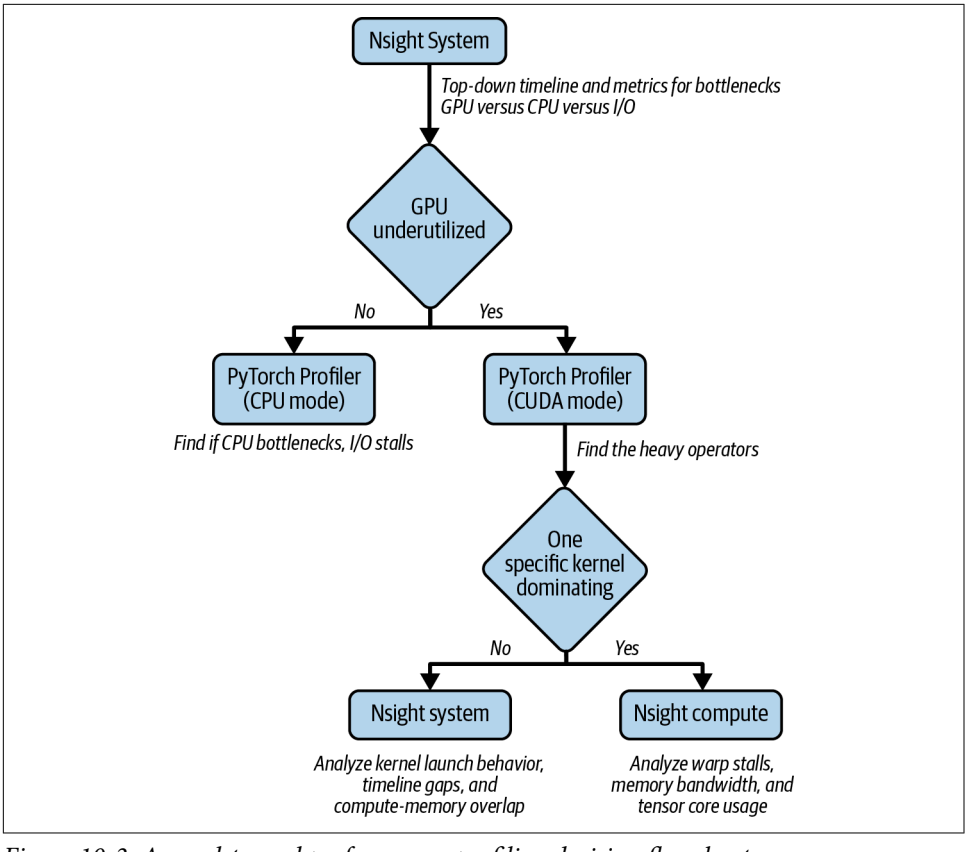

Figure 10-3. An end-to-end performance-profiling decision flowchart

Once you've identified a heavy operator, the question becomes whether a single kernel inside that operator is responsible for most of its time use. If one kernel dominates, use Nsight Compute to drill into the microarchitectural details, such as warp occupancy, memory stalls, and tensor-core utilization. These insights help you determine whether the kernel should be tuned, replaced, or fused. If no single kernel dominates, the issue is usually related to launch overhead, missing overlapping, or synchronization. In these cases, it is best to revisit the timeline in Nsight Systems.

Multimodal Serving

Another frontier lies in extending LLMs to multimodal models such as VLMs to process image, video, and audio. These models combine visual and textual reasoning, demanding serving infrastructures that can handle heterogeneous inputs, large image tensors, and cross-modal attention efficiently.

A key distinction to clarify is whether a model is *consuming multimodal input* or *producing multimodal output*. In this book, we only discuss language models that take in multimodal input but still generate text tokens using a standard autoregressive decoder.

Models that generate multimodal outputs such as images or video in applications like Midjourney, Sora, and Veo operate very differently. These systems typically rely on different generative architectures, such as diffusion-based models, as opposed to the autoregressive text generation discussed throughout this book. Because of that, we do not cover those approaches in this book.

Multimodal Input Processing

In this section, we'll use an example to show you how vision-based inputs are incorporated into language models. Following is a prompt that passes both an image and some text into the content section:

```
messages = [
   {
       "role": "user",
       "content": [
           {"type": "image", "image": image},
           {"type": "text", "text": "Describe this image."},
       ],
   }
]
```

After applying the prompt template, we can see the image section in the second row:

```
<|im_start|>user
<|vision_start|><|image_pad|><|vision_end|>
Describe this image.<|im_end|>
<|im_start|>assistant
```

Its input ID will be:

```
[151644, 872, 198, # <|im_start|>user
 151652, # <|vision_start|>
 151655, … (644 times) … , 151655, # placeholder for image embedding
 151653, # <|vision_end|>
 74785, 419, 2168, 13, 151645, 198, # Describe this image. <|im_end|>
 151644, 77091, 198] # <|im_start|>assistant\n
```

Finally, during model processing, the text token IDs are mapped to text embeddings as usual, while the image placeholder is replaced by embeddings from the Vision Encoder. It divides the image into small patches, then projects each patch into the LLM's hidden dimension, and processes them so that the patches can attend to one another and encode their global spatial and semantic relationships for the final vision embedding (Figure 10-4).

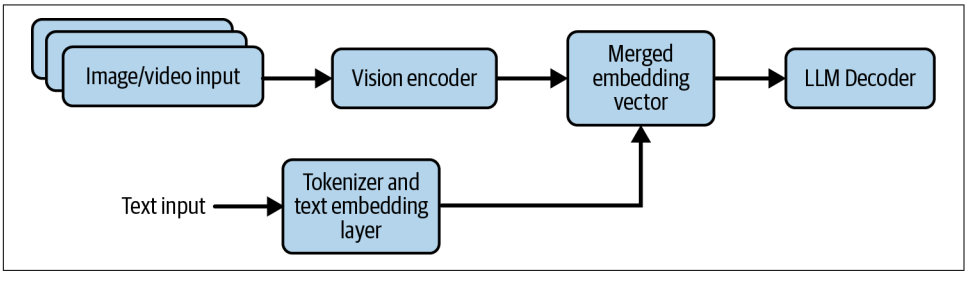

Figure 10-4. Dual-stream processing of text and vision inputs in a VLM

Architectural and System Implications

As multimodal models move into production, the principles of caching, batching, and streaming need to extend beyond text.

Text tokenization is relatively lightweight and usually does not cause significant CPU overhead. However, a key issue with serving these models is that when you have very fast GPUs run small models with higher batch sizes, the CPU overhead can become apparent.

On the contrary, integrating multimodal inputs introduces several computationally demanding, CPU-heavy tasks during preprocessing. These include converting high-resolution images to raw pixels, cropping and resizing, and complex tensor transformations that must complete sequentially before the LLM can begin. This requires front-loading the computational work, which causes an early CPU bottleneck. If the incoming request rate saturates the CPU's ability to prepare and feed these large tensors, the powerful GPU is left idle. Consequently, this resource imbalance shifts the primary constraint on performance. Now the overall request throughput becomes limited not by the LLM, but by the latency and efficiency of the initial, CPU-heavy vision-preprocessing stage.

vLLM's evolution between versions 0 and 1 provides a good example of how to mitigate the bottlenecks that multimodal inputs introduce. Version 1's core improvement involves adopting a completely asynchronous process to decouple CPU-intensive tasks from GPU execution. In V0, multimodal input preprocessing and API server overhead often blocked CPU operations, leading to GPU idle time. V1 offloads these tasks to separate CPU processes that run concurrently with the GPU's core inference loop. This decoupling minimizes CPU overhead, ensuring the GPU is

continuously fed data and operates at near-full utilization. The result is significantly higher throughput for both text and multimodal workloads.

Figure 10-5 illustrates how vLLM v1 uses two separate processes: Process 0 (top) and Process 1 (bottom), divided by the dotted line. Process 0 is responsible for the API server, input preprocessing (where most of the heavy multimodal work occurs), and output postprocessing. Process 1 is an entirely separate process dedicated to GPU kernel scheduling and launching. This separation ensures that even if Process 0 is occupied with expensive multimodal preprocessing, it does not block Process 1 from continuously launching GPU kernels, thereby reducing overall CPU overhead.

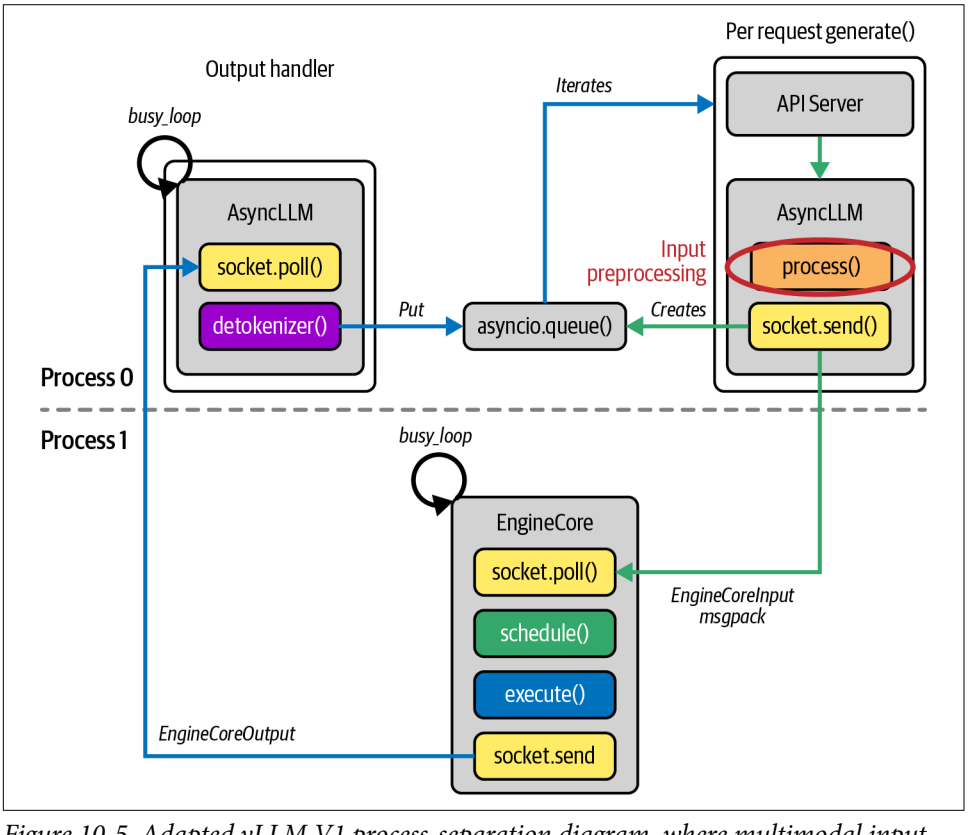

Figure 10-5. Adapted vLLM V1 process-separation diagram, where multimodal input preprocessing is separated as a nonblocking independent process (source)

Edge AI: Drivers and Enablers

While most of this book has focused on large-scale cloud serving, we note an important parallel trend: more and more AI workloads are being served on edge devices. This shift is driven primarily by three forces:

Latency
: For applications such as robotics, autonomous driving, and AR/VR, where latency is critical and data needs to be processed to react in milliseconds, even 20 to 50 ms network hops to the cloud are unacceptable. When computation moves to the edge, the data is processed immediately, enabling the real-time decision making that is essential for safety-critical and time-sensitive operations.

Data locality
: Processing data locally is essential in some cases, especially for raw audio or video and for sensitive data such as medical records, financial details, and proprietary industrial data. Transmitting such data to external cloud servers increases the risk of interception or unauthorized access and often conflicts with stringent data sovereignty and privacy regulations like GDPR and HIPAA. In these cases, using edge AI allows you to process sensitive data locally, on the device, so that raw data never leaves the premises.

Cost
: The sheer volume of data generated by modern devices, especially Internet of Things (IoT) sensors and high-resolution cameras, can overwhelm network infrastructure and drive up cloud costs. For example, transmitting live video streams to the cloud for processing is extremely expensive in terms of network bandwidth and incurs high cloud data transfer and storage costs. Edge AI allows data to be processed and filtered locally, sending only the resulting metadata, summarized insights, or necessary updates back to the cloud when needed. This significantly reduces bandwidth consumption and minimizes the reliance on continuous, high-volume data transfers to a centralized cloud infrastructure, translating to substantial cost savings.

Deploying AI at the edge is not simply a matter of shrinking a cloud model and pushing it onto a device. It is enabled by a convergence of hardware advances, model-compression techniques, software runtimes, and architectural patterns that make real-time intelligence feasible under tight power and memory budgets. The following enablers form the foundation that allows modern edge AI systems to operate efficiently and reliably.

Specialized Low-Power Hardware

Edge AI is only feasible because today's hardware can perform complex AI tasks with minimal power consumption. Only a few years ago, running deep-learning inference on a mobile device or embedded board meant dealing with slow CPUs, limited memory, and batteries that collapsed under sustained workload. To solve this, the industry pivoted toward *AI accelerators*, which are specialized chips or dedicated blocks within a *system on a chip* (SoC) designed to do one thing exceptionally well: high-speed, low-precision tensor math. The most significant development in this space is the *neural processing unit* (NPU). Unlike a CPU, which might have a few powerful cores, an NPU consists of thousands of tiny, efficient processing elements that operate in unison.

These local chips have very different target metrics from the cloud, where raw performance is king. At the edge, efficiency is the monarch. A server in a datacenter has access to virtually unlimited power and cooling, but a battery-operated, fanless edge device has a strict power budget. Consequently, the industry-standard metric for these accelerators is *tera-operations per second per watt* (TOPS/W). Modern accelerators can now deliver strong performance without draining the battery significantly or melting the casing.

Model Compression and Optimization

As Chapter 6 discussed extensively, even for cloud-served models, model compression is now the essential technique. To fit edge AI models onto edge devices with limited memory, model compression techniques have been even more important. While cloud servers use compression primarily to optimize throughput and maintain latency, edge devices often use it as a prerequisite for functionality. Without these techniques, modern deep-learning models simply cannot fit within the tight constraints of on-device SRAM or flash storage.

Along with the compression techniques we've discussed before, such as quantization, pruning and distillations, other model optimization techniques are tailored specifically for edge AI, such as KV caching, speculative decoding, and kernel-level optimization.

Heterogeneous Compute

Early edge AI ran entirely on one processor, resulting in slow, thermally constrained serving performance. Today's mobile and embedded platforms, like those from Apple, Qualcomm, and Intel, are fundamentally heterogeneous: each contains multiple specialized compute units designed for different portions of the workload. This type of serving is basically pipeline parallelism, similar to what we discussed in Chapter 7, but across different hardware and available cores. For example, the CPU

often performs preprocessing tasks such as image resizing, color space conversion, and input normalization. These operations involve branching logic and memory manipulation, areas where general-purpose cores excel. The NPU then executes the compute-dense layers of the neural network, taking advantage of its quantization-friendly, high-throughput matrix engines. Finally, the GPU may handle postprocessing tasks, such as overlaying bounding boxes or rendering augmented visuals in real time. Figure 10-6 illustrates this division of labor.

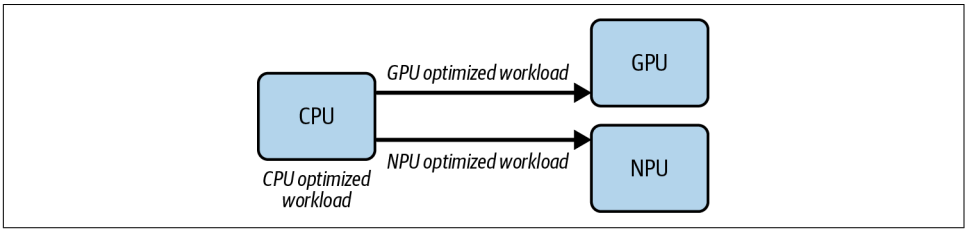

Figure 10-6. CPU, GPU, and NPU heterogeneous compute

More advanced runtimes even support *subgraph partitioning*, where individual layers within a single model are dynamically assigned to different accelerators to maximize throughput and minimize stalls. While this approach can deliver major performance gains, it introduces nontrivial challenges around synchronization, memory movement, and scheduling—making heterogeneous compute one of the most sophisticated enablers of modern edge AI systems.

Thermal-Aware Scheduling

In cloud computing, if a server gets hot, a fan spins faster. On a fanless edge device, such as a small sensor or a phone in a pocket, heat kills performance. When a chip gets too hot, it throttles and drastically slows down to prevent physical damage. In this case, intelligent schedulers are needed to constantly monitor the temperature in real time. If the temperature approaches or exceeds critical threshold levels, the scheduler can dynamically switch to a smaller, less accurate model or reduce the frame rate before the hardware throttles. When required, the scheduler can also migrate the workload from a hot main core to a cool little core for a few milliseconds, to allow heat to dissipate on the main core. This helps to maintain system stability over raw peak performance.

Edge–Cloud Hybrid Compute

Nowadays, AI workloads increasingly rely on devices and cloud servers to collaborate, instead of operating in isolation. Edge devices handle the tasks that demand immediate response and strict privacy, such as wake-word detection (for example, “Hey Siri”), lightweight image preprocessing, local video encoding, and running a tiny on-device LLM for rapid intent understanding. These operations take advantage

of NPUs and low-power GPUs while avoiding the latency and data exposure of sending raw inputs to the cloud. Once the input is filtered, features are extracted, and embeddings are compressed, the cloud executes the heavyweight components with large models. In this way, because the edge has preprocessed and condensed the input, the cloud consumes fewer resources and delivers faster overall performance, as shown in Figure 10-7.

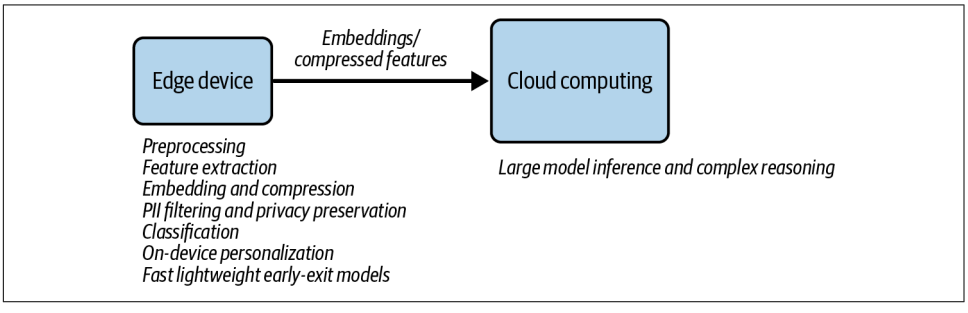

Figure 10-7. Edge–cloud hybrid inference architecture

More and more systems are using *adaptive offloading*, choosing dynamically whether to run a step on the device or in the cloud, based on bandwidth, battery, thermal conditions, or latency requirements. This cooperative setup takes advantage of the strengths of both environments to deliver a seamless user experience: real-time, private computation happens at the edge and large-scale intelligence is done in the cloud.

Multi-LoRA Serving

Next, we turn to serving fine-tuned models through *parameter-efficient fine-tuning* (PEFT). Rather than updating all of a model's parameters, PEFT adapts only a small subset of parameters while keeping the majority frozen, making fine-tuning faster and more efficient. The idea of serving models with fine-tuned lightweight adapters is not entirely new. However, as companies adopt LLMs in-house, they mostly focus on the first stage: incorporating proprietary or domain-specific knowledge to the model context, usually with RAG. The next natural step is fine-tuning, where teams aim for deeper alignment: improving accuracy, coherence, and grounding to match their unique data, workflows, and communication style.

LoRA has become one of the most popular methods in PEFT. It inserts small, trainable low-rank layers into the model's architecture, allowing it to learn task-specific adaptations without altering the original weights. The result is a faster, cheaper, and more modular way to fine-tune LLMs.

Muti-LoRA serving means loading multiple LoRA adapters into GPU memory so that different requests of LoRA adapters are served together on one model serving

instance. This request has two key points. First, all adapters that are active for serving requests are loaded in GPU memory ready to serve requests, as shown in Figure 10-8. They are not loaded one at a time. Inactive (or "cold") LoRA can be loaded from CPU memory or disk if needed.

Second, these requests still need to be batched with continuous batching to maintain high GPU utilization and high throughput, as you learned in Chapter 6. Because of that, specialized kernels such as the Punica kernel have been developed to process the calculation both with the main model weights and with multiple different specific adapters, then combine them all together.

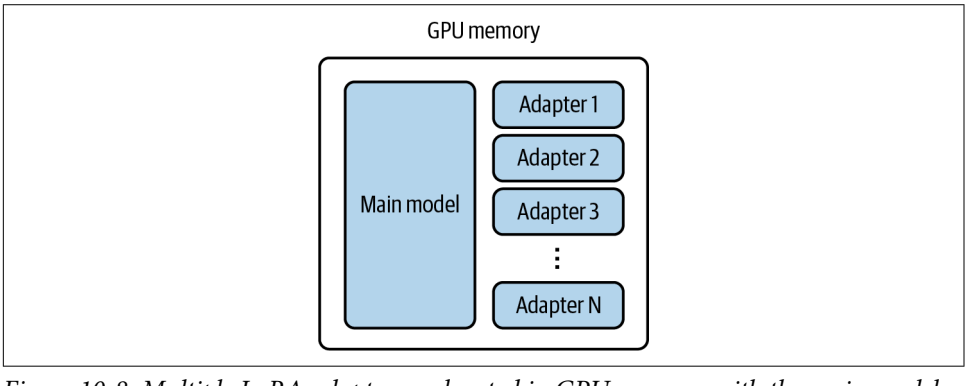

Figure 10-8. Multiple LoRA adapters co-located in GPU memory with the main model

The key benefit here is that a single base model instance can serve multiple fine-tuned LoRA adapters concurrently, reducing the number of GPUs needed (as shown in Figure 10-9), while still enabling domain-specific behavior or per-tenant customizations.

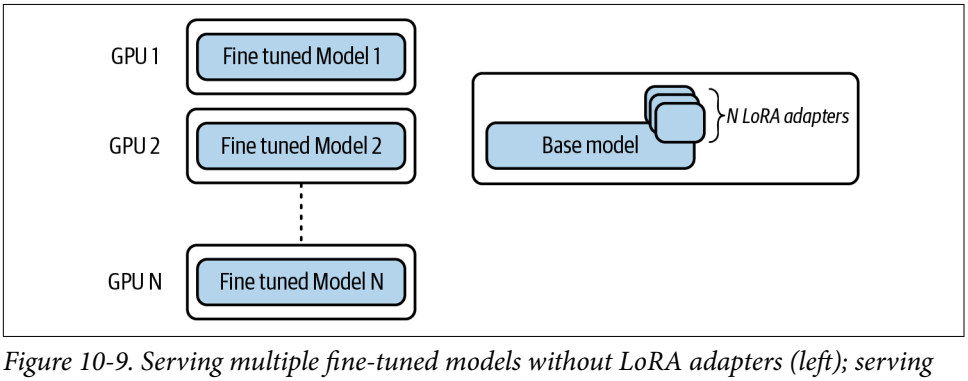

Figure 10-9. Serving multiple fine-tuned models without LoRA adapters (left); serving multiple fine-tuned models with N LoRA adapters, which only needs 1 GPU instead of N GPUs (right)

Now the question is: should you always always adopt this multi-LoRA serving setup for serving fine-tuned models? Not necessarily. Suppose that, in your use case, each of the fine-tuned LoRA adapters supports heavy traffic that requires multiple model replicas to scale horizontally (data parallelism). In this case, you'd be better off with merging each LoRA adapter into the main model and serving them independently. Multi-LoRA serving is primarily for serving LoRA adapters in cases where per-adapter traffic is relatively low and serving each adapter independently cannot saturate the hardware.

Model Serving in Reinforcement Learning

While PEFT methods like LoRA mostly focus on adapting models through supervised training, recent progress in reinforcement learning for LLMs, such as reinforcement learning from human feedback (RLHF), has opened a new frontier. Instead of simply training models on labeled data, RLHF incorporates human preferences to refine how the models respond. The final outcome is not just a model that knows the right answers, but one that is more helpful, polite, and safe—and aligns better with human intent. RL-based tuning is especially valuable in open-ended generation tasks, where correctness is very subjective. Figure 10-10 shows where reinforcement learning sits in the stages of LLM training.

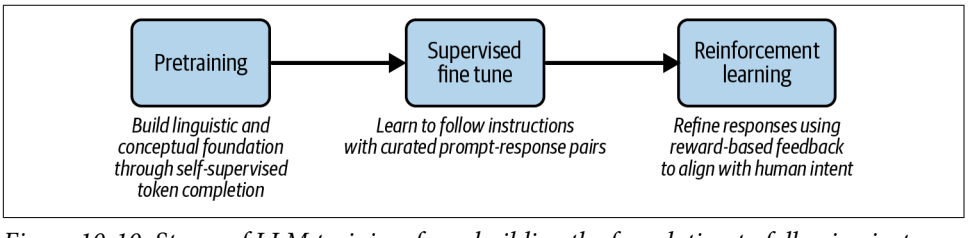

Figure 10-10. Stages of LLM training, from building the foundation to following instructions to aligning with humans

LLM Serving in RL

OpenRLHF, a very popular open source RLHF framework, estimates that 80% of RLHF training time is used at the sample-generation stage of model serving. As shown on the left side of Figure 10-11, serving engines such as vLLM and SGLang play a critical role. In this phase, prompts are sent to the *actor model*, which is the current version of the language model being optimized during RLHF. In reinforcement learning terms, the actor model represents the "current policy."

As training progresses, the actor model is updated repeatedly, and the "current policy" simply refers to its latest set of parameters. The actor model is deployed across multiple replicas using a high-throughput serving system. The goal is to efficiently generate large batches of candidate responses for downstream evaluation. These

generated responses are then passed to the training phase, shown on the right in Figure 10-11, where the reward model assigns quality scores and the reference model helps compute policy gradients. The actor model is updated based on these gradients, and its new weights are periodically synchronized back to the serving replicas.

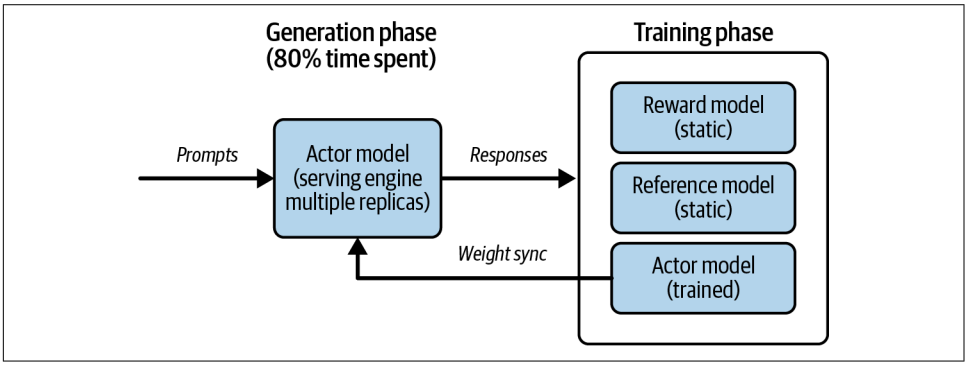

Figure 10-11. Simple reinforcement learning flow in the generation (serving) phase (left) and training phase (right)

This architecture highlights how serving systems, despite originally having been built for inference, are now an integral part of the RLHF training loop. They enable scalable, low-latency sampling from large models across thousands of concurrent requests.

Determinism in RL Serving

In these serving engines, determinism is just as critical as throughput and scalability. In RLHF, even minor nondeterminism across replicas or runs can propagate into inconsistent rewards, unstable training, and unreproducible results. In other words, *reproducible inference* is just as important as performance.

Recent work, such as Thinking Machines' "Defeating Nondeterminism in LLM Inference", shows that subtle implementation details such as batch-shape variation can introduce tiny deviations that accumulate over thousands of RLHF iterations. In pipelines that combine generation, reward scoring, and policy updates, such drift can destabilize rewards or gradient estimates. The modern serving engines used for reinforcement learning therefore focus not only on serving performance but on batch-invariant deterministic inference to ensure reproducible token outputs.

Summary

This chapter covered some of the most forward-looking techniques in LLM serving: semantic-aware caching and routing, system-level and kernel-level profiling, multimodal inference, scalable multi-LoRA deployment, and reinforcement-learning-

driven serving architectures. These topics reflect how quickly the field is expanding beyond pure text and into richer, more dynamic, and more personalized experiences.

With this, we bring the book to a close. Our goal has been to demystify the internals of high-performance LLM serving in a way that feels approachable rather than overwhelming. If this book has helped you see LLM serving not as a black box, but as an exciting engineering landscape to explore, then it has achieved its purpose.

The LLM serving space will continue to move very quickly. Hardware architectures will change, model families will grow, and new modalities will emerge, but the principles you've learned here will remain. Thank you for joining this journey. The tools are in your hands now, and the next generation of intelligent systems is yours to build.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

About the Authors

Chi Wang is a director of engineering at Salesforce's Einstein AI group, with over 18 years of experience in artificial intelligence and distributed systems. He leads the development of large-scale AI platforms that enable model training, inference, and optimization for hundreds of internal teams and power AI capabilities used by millions of Salesforce customers.

At Salesforce, Chi oversees multiple engineering teams focused on model inference and optimization, and data science platforms. His work spans building multi-tenant AI infrastructure, scaling distributed compute systems, and improving the performance and cost-efficiency of large language model workloads in production.

Chi is the lead inventor on 12 patents across areas including model serving and optimization, data access control, and large-scale system design. He is also a passionate technical writer, focused on making complex AI systems practical and accessible for engineers.

Peiheng Hu is an accomplished machine learning engineer with over 10 years of industry experience and expertise in building large-scale AI systems. He currently works at NVIDIA, where he focuses on the cutting-edge distributed LLM inference, pushing the boundaries of high-performance inference engines on the latest NVIDIA GPUs. He holds a master of science in computational science and engineering from Harvard University and a bachelor of science in industrial engineering operations research from Georgia Institute of Technology.

Previously, Peiheng served as a principal member of technical staff at Salesforce, where he led the development of the company's only unified serving platform, handling thousands of per-tenant models and LLM optimizations for Agentforce that saved millions in AI infrastructure expenses. Prior to that, he was a senior ML engineer at Microsoft Azure, where he architected distributed ML processing solutions for cloud security detection and analytics, handling billions of transactions per hour.

Colophon

The animal on the cover of *Hands-On LLM Serving and Optimization* is a dwarf lemur, of the genus *Cheirogaleus*. *Cheirogaleus* consists of 10 species, all of which live in the coastal forests of eastern Madagascar.

These small primates are 7 to 10 inches long, with a 6.5-inch tail. Their heads are rounder than the larger lemurs, but their muzzles are more pointed. They have gray-brown or reddish fur, depending on the species, with a white belly. They all have small ears, large eyes surrounded by dark rings, and long hind limbs.

Dwarf lemurs are nocturnal, arboreal quadrupeds. They are typically solitary, but sometimes live in pairs. They nest in tree hollows and only rarely spend time on the forest floor. Their diet consists of fruits and flowers. During the winter season, they hibernate and subsist on fat stored at the base of their tails.

The IUCN status for these species ranges from vulnerable to critically endangered. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by José Marzan Jr., based on an antique line engraving from *Natural History of Animals*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.