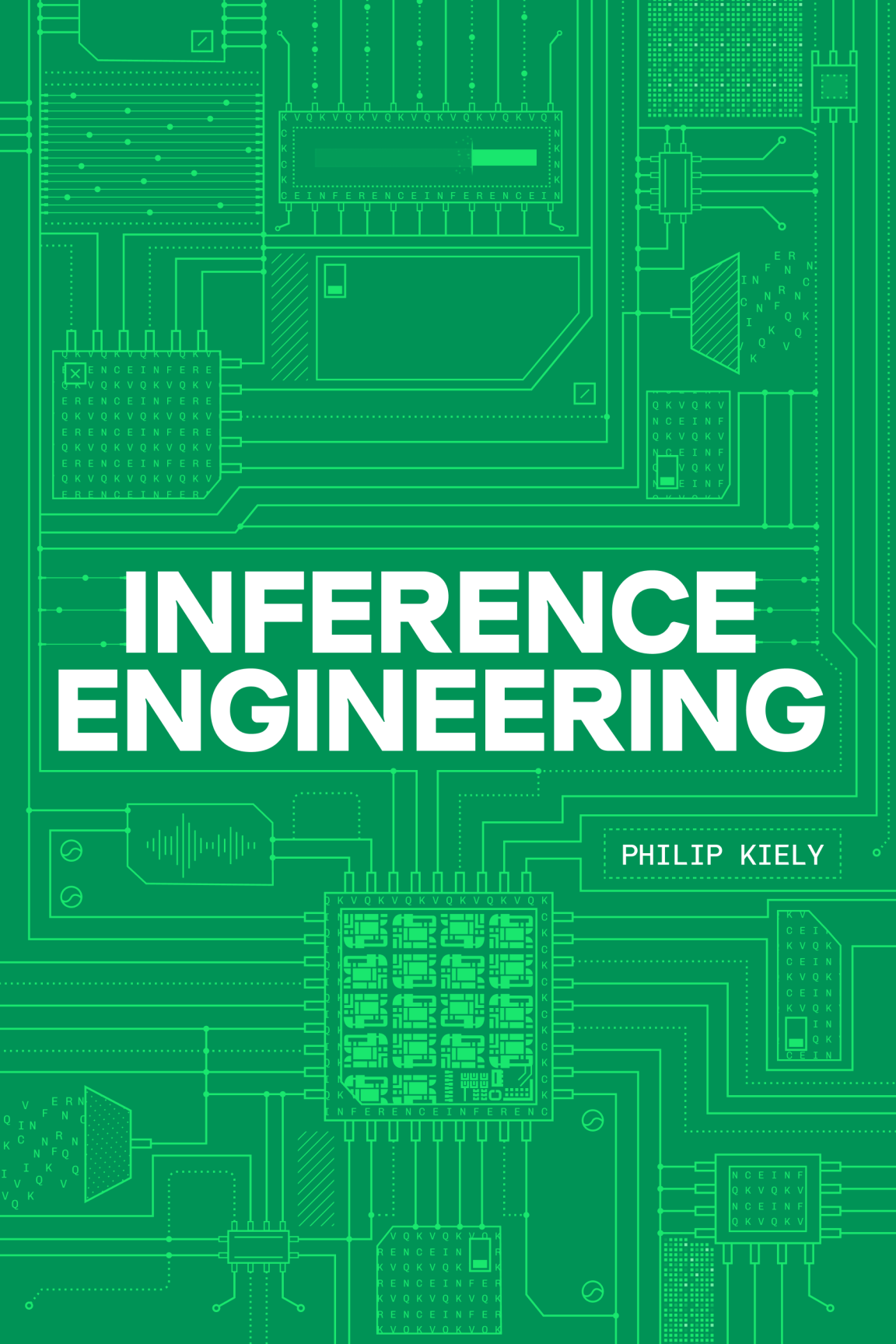

INFERENCE
ENGINEERING
PHILIP KIELY

INFERENCE ENGINEERING

BY PHILIP KIELY

Inference Engineering by Philip Kiely
Published by Baseten Books
Copyright © 2026 Baseten Labs, Inc

All rights reserved. No portion of this book may be reproduced in any form without written permission from the publisher, except as permitted by U.S. copyright law. For permissions contact Baseten.

Cover art by Luke de Haas
Edited by Robin Bourjaily
ISBN: 979-8-9943597-2-3

Table of Contents

Preface 9

Chapter 0: Inference 15

Chapter 1: Prerequisites 23

1.1 Scale and Specialization 26

1.2 About Your App 27

1.2.1 AI-Native Applications 28

1.2.2 Online versus Offline 29

1.2.3 Consumer versus B2B 30

1.3 Model Selection 31

1.3.1 Model Evaluation 31

1.3.2 Fine-Tuning for Domain-Specific Quality 32

1.3.3 Distillation 33

1.4 Measuring Latency and Throughput 35

1.4.1 Latency Percentiles 36

1.4.2 End-to-End Metrics 37

Chapter 2: Models 39

2.1 Neural Networks 42

2.1.1 Linear Layers and Matmul 44

2.1.2 Activation Functions 44

2.2 LLM Inference Mechanics 46

2.2.1 LLM Architecture 49

2.2.2 Transformer Blocks 50

2.2.3 Attention 52

2.2.4 Mixture of Experts Models 53

2.3 Image Generation Inference Mechanics 55

2.3.1 Image Generation Model Architecture 57

2.3.2 Few-Step Image Generation Models 59

2.3.3 Video Generation 59

2.4 Calculating Inference Bottlenecks 61
2.4.1 Ops:Byte Ratio and Arithmetic Intensity 62
2.4.2 LLM Inference Bottlenecks 63
2.4.3 Image Generation Inference Bottlenecks 67
2.5 Optimizing Attention 67

Chapter 3: Hardware 71
3.1 GPU Architecture 74
3.1.1 Compute 74
3.1.2 Memory and Caches 76
3.2 GPU Architecture Generations 77
3.2.1 Hopper GPUs 79
3.2.2 Ada Lovelace GPUs 80
3.2.3 Blackwell GPUs 81
3.2.4 Rubin GPUs 81
3.2.5 Grace and Vera CPUs 82
3.3 Instances 83
3.3.1 Multi-GPU Instances 84
3.3.2 Multi-Instance GPUs 86
3.4 Other Datacenter Accelerator Options 87
3.5 Local Inference 89
3.5.1 Desktop Inference 90
3.5.2 Mobile Inference 91

Chapter 4: Software 93
4.1 CUDA 96
4.1.1 CUDA Kernels for Inference 98
4.1.2 CUDA Kernel Selection 98
4.1.3 Reducing Memory Accesses with Kernel Fusion 100
4.2 Deep Learning Frameworks and Libraries 101
4.2.1 PyTorch 102
4.2.2 Model File Formats 103
4.2.3 ONNX Runtime and TensorRT 104
4.2.4 Transformers and Diffusers 105
4.3 Inference Engines 105
4.3.1 vLLM 106

4.3.2 SGLang **108**
4.3.3 TensorRT-LLM **109**
4.4 NVIDIA Dynamo **111**
4.5 Performance Benchmarking and Load Testing **112**
4.5.1 Performance Benchmarking Tooling **113**
4.5.2 Performance Benchmarking Tips **114**
4.5.3 Profiling Performance **114**

Chapter 5: Techniques 117
5.1 Quantization **120**
5.1.1 Number Formats **121**
5.1.2 Quantization Approaches **125**
5.1.3 Measuring Quality Impact **128**
5.2 Speculative Decoding **129**
5.2.1 Draft-Target Speculative Decoding **131**
5.2.2 Medusa **132**
5.2.3 EAGLE **133**
5.2.4 N-gram Speculation and Lookahead Decoding **134**
5.3 Caching **136**
5.3.1 Prefix Caching and KV Cache Re-Use **136**
5.3.2 Where to Store the KV Cache **139**
5.3.3 Cache-Aware Routing **140**
5.3.4 Long Context Handling **141**
5.4 Model Parallelism **142**
5.4.1 Tensor Parallelism for Lower Latency **144**
5.4.2 Expert Parallelism for Higher Throughput **145**
5.4.3 Multi-Node Inference **146**
5.5 Disaggregation **148**
5.5.1 How Disaggregation Works **148**
5.5.2 When to Use Disaggregation **149**
5.5.3 Dynamic Disaggregation with NVIDIA Dynamo **150**

Chapter 6: Modalities 153
6.1 Vision Language Models **156**
6.1.1 Video Processing for Vision Language Models **158**
6.1.2 Omni-Modal Models **159**

6.2 Embedding Models 159
6.2.1 Embedding Model Architecture 160
6.2.2 Embedding Model Inference 161
6.3 ASR Models 162
6.3.1 Single-Chunk Latency Optimization 163
6.3.2 Long File Latency Optimization 164
6.3.3 Diarization 166
6.4 TTS Models 166
6.4.1 Streaming Real-Time Text to Speech 168
6.4.2 Speech-to-Speech Models 168
6.5 Image Generation Models 169
6.5.1 Image Generation Kernel Optimization 170
6.5.2 One Weird Trick for Faster Image Generation 172
6.6 Video Generation Models 173
6.6.1 Attention Optimization and Quantization 174
6.6.2 Context Parallelism 176

Chapter 7: Production 177
7.1 Containerization 179
7.1.1 Dependency Management 181
7.1.2 NIMs 183
7.2 Autoscaling 183
7.2.1 Concurrency and Batch Sizing 186
7.2.2 Cold Starts 188
7.2.3 Routing, Load Balancing, and Queueing 190
7.2.4 Scale to Zero 192
7.2.5 Independent Component Scaling 192
7.3 Multi-Cloud Capacity Management 193
7.3.1 GPU Procurement 195
7.3.2 Geo-Aware Load Balancing 196
7.3.3 Building for Reliability 196
7.3.4 Security and Compliance 198
7.4 Testing and Deployment 199
7.4.1 Zero-Downtime Deployment 200
7.4.2 Cost Estimation 201
7.4.3 Observability 203
7.5 Client Code 204

7.5.1 Client Latency Overhead **205**
7.5.2 Asynchronous inference **205**
7.5.3 Streaming and Protocol Support **205**
7.6 Production Inference with Baseten **208**

Appendix A: Inference Glossary **209**
Appendix B: Recommended Reading **231**
Architecture **234**
Developer Tools **239**
Frontier Open Models **243**
GPU Infrastructure **246**
Inference Optimization Research **249**
Intelligence Evaluation **254**
Acknowledgements **257**

Preface

Inference is the most valuable category in the AI industry.

Inference engineering, on the other hand, is still in its infancy. Inference engineers work across the stack from CUDA to Kubernetes in pursuit of faster, less expensive, more reliable serving of generative AI models in production.

On November 30, 2022 – the day that ChatGPT was launched – there were perhaps a few hundred inference engineers in the world, though they didn't call themselves that at the time. These specialists mostly worked at frontier labs like OpenAI, Midjourney, and Anthropic or big tech companies like Google and NVIDIA.

Back then, it looked like this might be the way of the AI industry. Perhaps training generative AI models would be so hard, and so expensive, that only a handful of companies would develop closed models and thus require inference engineering for production serving. In this alternate future, the rest of the world would be mere consumers of AI via APIs, renting intelligence a token at a time.

Three years later, it turns out that training generative AI models is hard. It is expensive. But it is neither so hard nor so expensive that it is limited to that handful of players.

Instead, a Cambrian explosion of open models – more than two million and counting on Hugging Face – means that every engineer can now deploy their own intelligence to power their AI products. Research labs around the world, from OpenAI and NVIDIA Nemotron in America to Mistral AI and Black Forest Labs in Europe to Alibaba Qwen, DeepSeek AI, Z AI, and Moonshot AI in China, regularly release open models of all modalities.

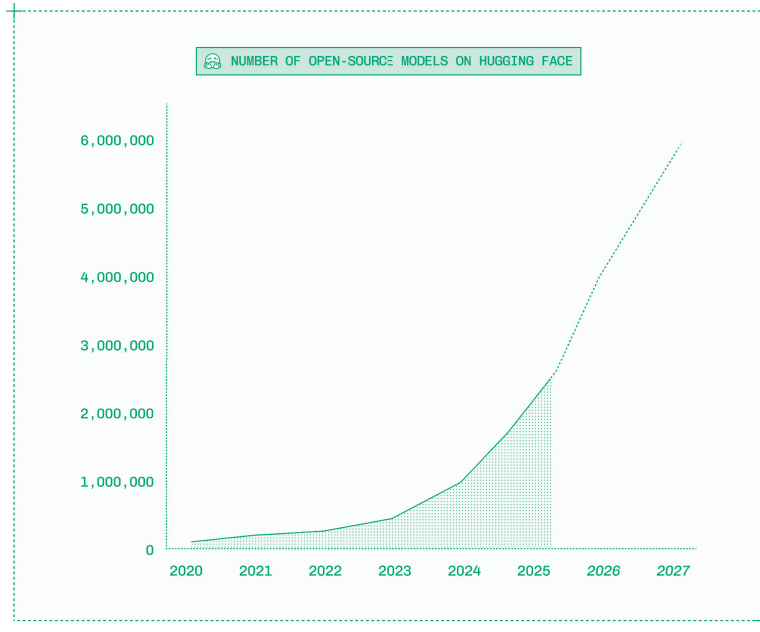


Figure P.1: There are well over two million open models on Hugging Face, 25 times more models than five years ago.

Despite closed models getting smarter and cheaper, the movement into open models is accelerating. Open models differ in the availability of their weights:

- **Closed model:** A proprietary model where weights are unavailable to the public, like GPT-5 or Claude Sonnet.
- **Open model:** A model where weights are publicly available, like Llama or DeepSeek, usually released under the MIT license or a similar permissive license (though some models restrict commercial use, always double-check license terms).

Until December 2024, there was a meaningful gap in intelligence between closed and open models. When DeepSeek V3 and R1 were released, that gap disappeared.

Today, new closed models are matched by open models within months if not weeks, with occasional open models like Kimi K2 Thinking even exceeding closed model capabilities for brief windows.

Even if open models are constantly chasing closed models on benchmarks, they still change the equation for AI product builders. As both types of models get better, closed and open models cross capability thresholds and power new classes of products.

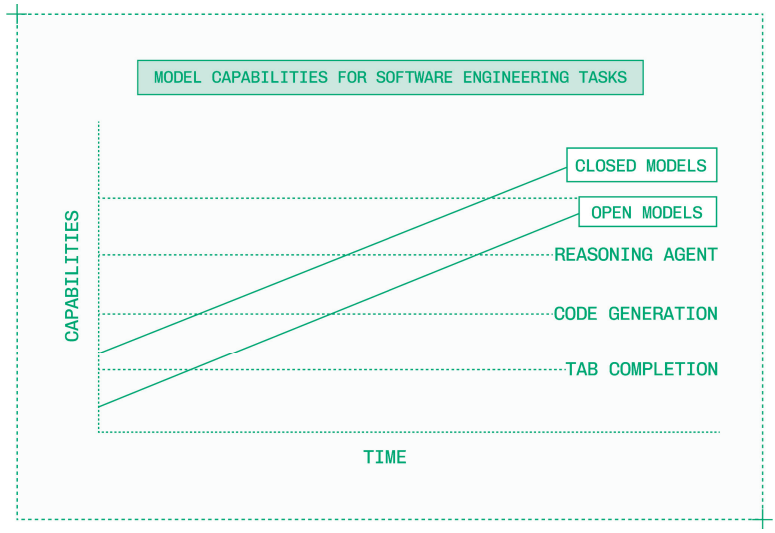


Figure P.2: Open and closed models are both improving rapidly, unlocking and expanding access to new capabilities.

In 2022, it was impossible to build the kinds of AI-native products that define the industry today.

Over time, closed models got smarter and new categories like customer service voice agents and AI-powered IDEs became possible. These early models were slow, expensive, and unreliable, but the capabilities were there and AI engineers began building companies around these capabilities.

As open models crossed the same capability thresholds, these product builders began using them to replace closed models. Many also

began fine-tuning open models to cross capability thresholds faster and even exceed closed model quality for their specific product and domain.

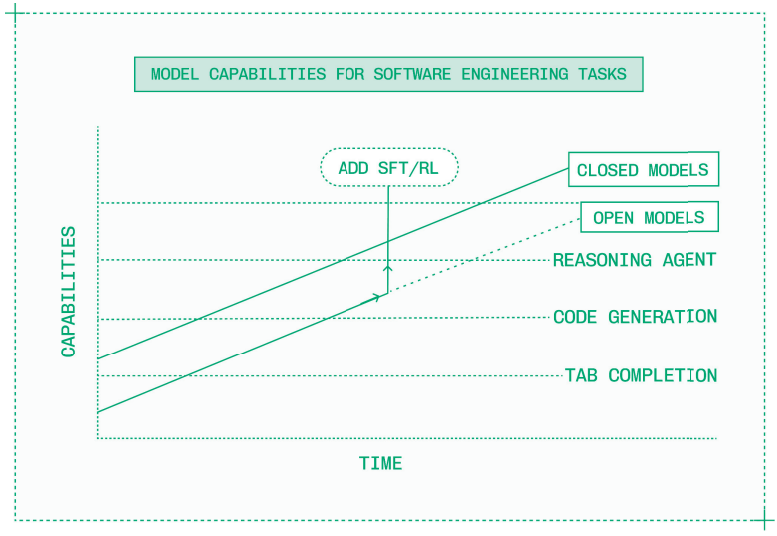


Figure P.3: Customizing open models unlocks new capabilities while retaining control over latency, reliability, and economics.

Switching to open models unlocks the opportunity to use inference engineering to make the models powering AI products better in new dimensions:

- **Latency:** Closed model APIs are built for throughput, but open models can be optimized for real-time applications.
- **Availability:** While APIs for GPT and Claude are stuck at two nines of uptime, it's possible to achieve four nines or better with dedicated deployments of open models.
- **Cost:** Open models are often at least 80 percent less expensive at scale.

So where three years ago it looked like inference engineering might be a niche field, today every company aiming to build truly differentiated and competitive AI products needs an inference strategy.

AI-native startups like Cursor, Clay, Gamma, and Mercor are redefining hypergrowth building products that rely on open and in-house models. Leading digital native companies like Notion and Superhuman are thriving by deeply integrating AI capabilities into products that hundreds of millions already love.

And a new generation of blended research and engineering teams – World Labs, Writer, Mirage, and dozens of others – are building enormous businesses by training and productizing their own foundation models.

Adoption is even strong in enterprise and regulated industries, which historically have been slow to adapt to new technologies. Companies like OpenEvidence, Abridge, and Ambience are making generative AI ubiquitous in healthcare, while at the world's largest companies, AI initiatives are moving past the pilot stage into massive user adoption.

I've been incredibly fortunate to have a front-row seat to the fastest-moving market in history over the last four years at Baseten, where we power mission-critical inference for the best AI products, including every company listed in the previous paragraphs.

The incredible market-wide demand for inference means that everyone from developers to executives has the opportunity to learn inference engineering and use it to advance their career and business.

You are early. While the potential and impact of inference are becoming clear, the space is young. There are relatively few people working on inference, and newcomers can become experts quickly. There are enormous opportunities to solve novel, interesting, and deeply technical problems at all levels of the stack.

Inference Engineering is your guide to becoming an expert in inference. It contains everything that I've learned in four years of working at Baseten. This book is based on interviews with dozens of experts from our engineering team; technical talks I've delivered at conferences like NVIDIA GTC, PyTorch Conference, AWS re:Invent, and AI Engineer World's Fair; and countless conversations with customers and builders around the world.

Thank you for reading *Inference Engineering* and welcome to the early days of inference.

Philip Kiely
San Francisco, CA

CHAPTER 0

Inference

Inference

Inference is the second phase in a generative AI model's lifecycle:

- **Training:** The process of learning model weights from data.
- **Inference:** Serving generative AI models in production.

In last decade's machine learning (ML) boom, hundreds of thousands of data scientists and ML engineers became familiar with the full lifecycle, both training and inference, for ML models.

Inference for classic ML models is relatively straightforward. In the early days of Baseten, we ran inference for models built with tools like XGBoost on lightweight CPUs with a simple software stack.

In contrast, inference for generative AI models is complex. You can't simply take model weights, get some GPUs, and expect inference to be fast and reliable enough for large-scale production use. Doing inference well requires three layers:

- **Runtime:** Optimizing the performance of a single model on a single GPU-backed instance.
- **Infrastructure:** Scaling across clusters, regions, and clouds without creating silos while maintaining excellent uptime.
- **Tooling:** Providing engineers working on inference with the right level of abstraction to balance control with productivity.

These three layers must work together to create a system that can handle mission-critical inference at scale.

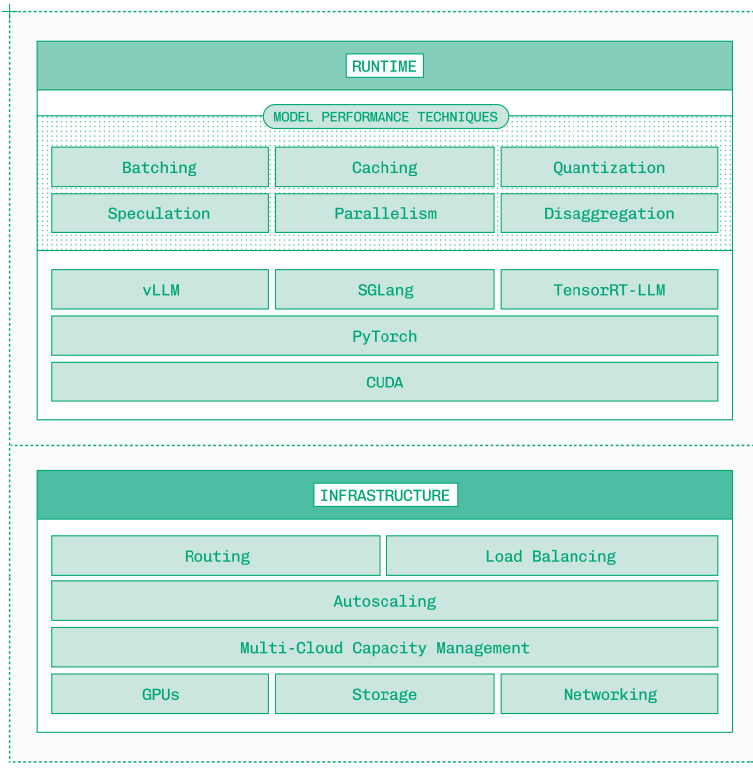


Figure 0.1: A complete inference stack includes both runtime and infrastructure optimizations.

The runtime layer is responsible for ensuring that an individual model running on a GPU (or across several GPUs in a single instance) is running as performantly and efficiently as possible.

This layer depends on a sophisticated software stack, from CUDA to PyTorch to inference engines like vLLM, SGLang, and TensorRT-LLM. Low-level optimization is important, with kernels like FlashAttention delivering significant performance gains.

The runtime layer relies on a number of model performance techniques that apply new research to the unique challenges of inference on generative AI models:

- **Batching:** Run incoming requests in parallel, weaving them together on a token-by-token basis to increase throughput.
- **Caching:** Re-use the KV cache – the cached results of the attention algorithm – between requests that share prefixes.
- **Quantization:** Lower the precision of select pieces of the model to access more compute and reduce memory burden.
- **Speculation:** Generate and validate draft tokens to produce more than one token per forward pass during decode.
- **Parallelism:** Efficiently leverage more than one GPU to accelerate large models without introducing new bottlenecks.
- **Disaggregation:** Separate the two phases of LLM inference, prefill and decode, onto independently scaling workers.

These model performance techniques are used for models of all modalities, not just LLMs. Modalities like vision language models, embedding models, automatic speech recognition, speech synthesis, image generation, and video generation extend the capabilities of AI systems and require their own inference optimizations.
These runtime optimizations are not enough. No matter how performant a single instance of a model server is, it will eventually receive more traffic than it can handle.

This is not a CUDA problem or a PyTorch problem. This is a systems problem that needs to be solved at the infrastructure layer.

The nature of infrastructure problems changes with each level of scale. At first, the problems are around autoscaling: knowing when to add and remove replicas and figuring out how to do so quickly.

Past a certain scale, generally a few hundred GPUs, infrastructure problems are defined by capacity. To get access to enough GPUs, inference engineers begin spreading workloads across multiple regions and multiple cloud providers.

This quickly leads to silos, where models in one cluster may be starved for resources while other clusters have unused capacity. The final level of scale in infrastructure is a global system that treats all available resources as a single unified pool of compute.

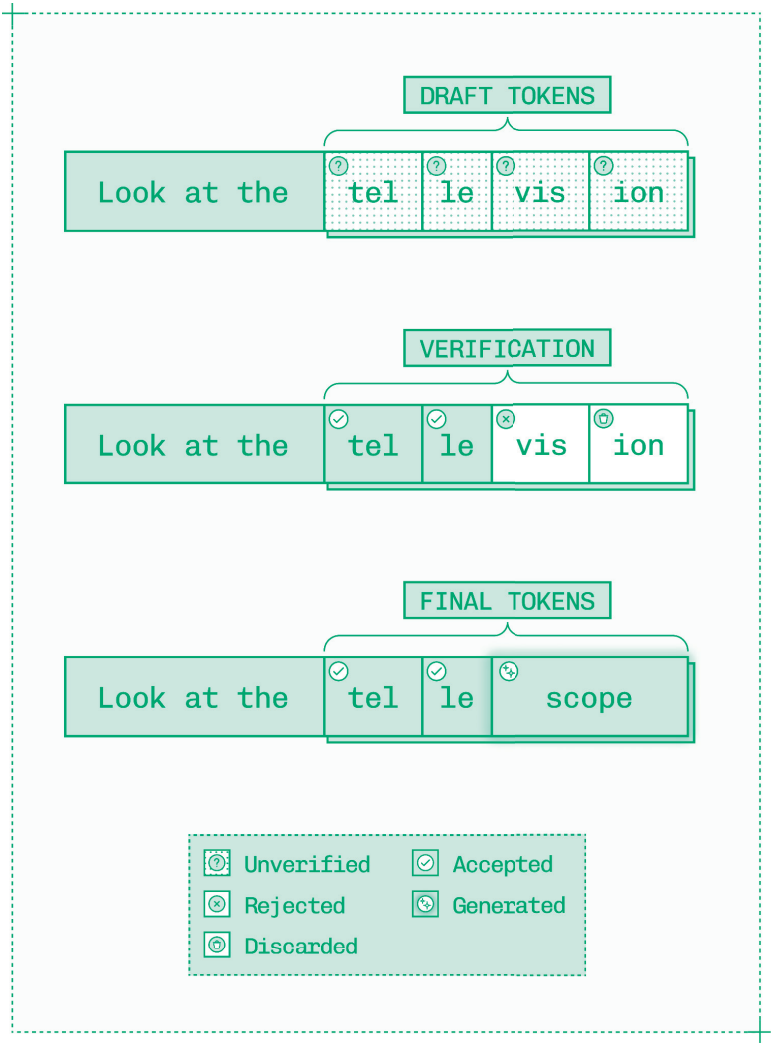


Figure 0.2: Runtime performance optimizations like speculation, diagrammed here, improve inference latency.

Thoughtful multi-cloud infrastructure also improves reliability, protecting against downtime in any individual region or cloud provider. And for global applications, running inference near end users improves end-to-end latency.

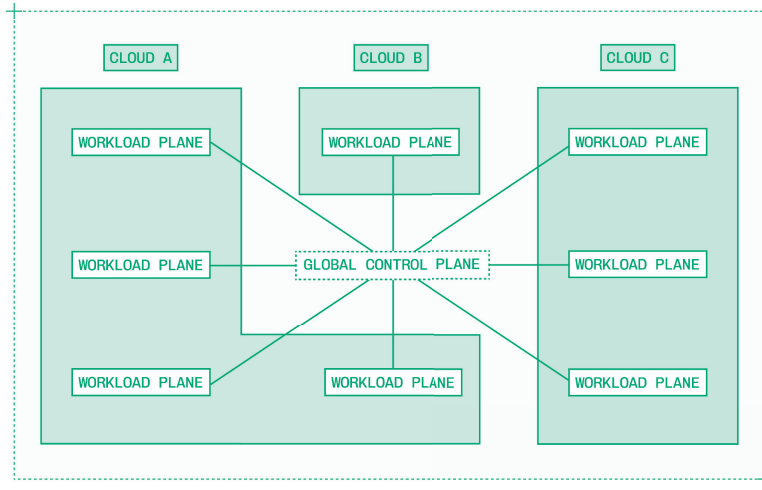


Figure 0.3: Serving model inference at scale requires unifying capacity across multiple cloud service providers.

Once these runtime and infrastructure capabilities are built, they need to be presented at the appropriate level of abstraction. Both inference providers like Baseten and internal teams building inference need to consider what tooling and developer experience to provide as the critical third layer in a complete inference platform.

Developer experience is subjective. For inference, one extreme is the black box: give a platform model weights, get back an API. At the other extreme is providing only basic constructs for compute, network, disk, and so forth.

The right developer experience is somewhere in the middle, where inference engineers have enough control to run mission-critical inference confidently, but enough abstraction to work productively.
Inference Engineering presents a map of the technologies and techniques that power inference across all three layers of runtime, infrastructure, and tooling.

Chapter 1, Prerequisites, covers the product thinking and AI engineering work that need to be done before inference engineering comes into play: use case definition, latency and cost budgeting, and selecting and evaluating which generative AI models to optimize and deploy.

Chapter 2, Models, introduces the technical architecture of AI models – from large language models to image and video generation models – and establishes where the bottlenecks exist for inference with a special focus on optimizing attention.

Chapter 3, Hardware, starts at the spec sheet for modern GPUs and breaks down compute and memory, then disambiguates architectures and SKUs within NVIDIA's datacenter-grade offerings before briefly surveying other accelerators on the market.

Chapter 4, Software, builds abstractions from CUDA to frameworks like PyTorch, Transformers, and Diffusers and inference engines like vLLM, SGLang, and TensorRT-LLM. It also introduces Dynamo, NVIDIA's latest system for large-scale distributed model serving.

Chapter 5, Techniques, discusses key model performance optimization techniques adapted from cutting-edge research and applies them in production: quantization, speculative decoding, KV cache re-use, model parallelism, and disaggregation.

Chapter 6, Modalities, expands inference engineering beyond LLMs to voice and visuals. Many types of generative AI models – vision-language models, embedding models, automatic speech recognition (ASR) models, and speech synthesis models – adapt LLM architectures, meaning inference engineers can run them with the same tools and techniques used with LLMs. Image and video generation models have their own architectures and associated performance optimization techniques.

Chapter 7, Production, concludes the book with a rundown of the important problems to solve in operating infrastructure for and building performant applications on optimized model inference services.

Appendices A and B add a glossary of inference engineering terms and a collection of recommended resources for further reading, respectively.

Like LLMs, books have knowledge cutoffs. This book was finished in January 2026. While details will change, the principles, concepts, and foundational technologies in this book provide a strong background on inference engineering that will serve you well for years to come.

CHAPTER 1

Prerequisites

Prerequisites

Inference engineering adds speed and scale to AI products by optimizing production serving of generative models.

Optimization means identifying the best solution out of a range of options. Before optimizing model performance and building robust infrastructure, you need to know what "best" means for your product – many performance improvements come from making tradeoffs among latency, throughput, and quality.

In practice, optimization is often about finding the right balance rather than maximizing a single factor.

NFL players are big, fast, and strong. But they're not as big as sumo wrestlers, as fast as Olympic sprinters, or as strong as champion powerlifters. Their bodies and skills are optimized to fulfill the specific demands of their position over the course of a full season.

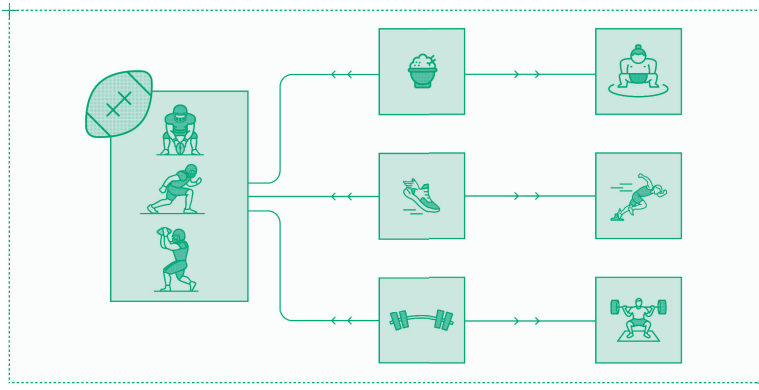


Figure 1.1: Just like elite athletes, inference services must be specialized for the demands of their workloads.

Similarly, your inference system must be optimized to fulfill the specific demands of your model, your product, and your traffic. The more constraints you can introduce, the better outcomes you can achieve.

You should know your:

- **Model requirements:** Which model(s) do you need to run inference on?
- **Application interface:** How will inputs be delivered to the model, and how is the output expected to be formatted?
- **Latency budget:** End-to-end, how fast does your product need to respond to a user action?
- **Unit economics:** What does it make sense to spend on a per-request, per-user, or per-month basis?
- **Usage patterns:** How many concurrent users are you serving, and is there any pattern to their usage (e.g., more activity during business hours)?

Early on in building an AI product, the answers to these questions may not be clear. At this early stage, it's often better to use off-the-shelf APIs whenever possible rather than investing in dedicated inference. As the product scales, requirements become clear and inference engineering becomes a worthwhile pursuit.

1.1 Scale and Specialization

There are two ways that you can add AI models to your product:

- **Shared inference:** Send your traffic to a public API endpoint for a given model and pay per million tokens or some other consumption-based metric.
- **Dedicated deployments:** Rent GPUs and set up an inference service exclusively for your application, paying per hour of GPU time (or purchase and install GPUs on-premises).

Shared versus dedicated inference is not exactly the same conversation as closed versus open models – there are plenty of shared endpoints for open models and many model labs offer large customers some kind of dedicated setup for their closed models. However, one of the key motivations for adopting open models is that it unlocks unrestricted dedicated inference.

Most AI products start with pay-per-token APIs because the tradeoffs make sense while looking for product-market fit or in the early stages of growth.

Pros of shared inference	Cons of shared inference
Zero overhead, only pay for consumption	Cost scales linearly with usage
No cold start times, model is always available	Provider uptime caps product uptime, noisy neighbors
Minimal engineering work, just need an API key	No control over latency, model quality, or rate limits

Over time, shifting AI products to dedicated deployments is essential for three reasons:

- **Scale:** You are processing enough volume of traffic that it's more economical to pay per GPU than per million tokens.
- **Specialization:** You are running a custom or fine-tuned model, or you have specific latency or uptime requirements.
- **Orchestration:** Your product relies on multiple models and multi-step pipelines and you need to minimize network latency and deployment complexity.

The switch to dedicated deployments puts you in charge of your own inference engineering. This gives you flexibility and control but adds to your engineering surface area and increases the floor of your monthly spend on inference. Only switch once there is a clear and immediate business need.

1.2 About Your App

Every inference engineering decision you make will be downstream of your use case.

Imagine a sports coach recruiting at a school. Which students would they want to talk to? It would depend entirely on which sport they're coaching – the basketball coach would want the tallest kids in the class, but the gymnastics coach would select among the shortest.

Similarly, the way your inference system will be used determines how you go about building it.

There are two cases where inference engineers need to build at the highest level of generality:

- **Foundation models:** You have trained your own model and will sell consumption directly via a public shared inference API and need to support many usage patterns.
- **Inference platforms:** You are building an inference platform, either internally or as a product, and need to support any model and any use case.

But most AI-native applications are vertical apps like code editors or customer service agents, where AI is used to create some novel user experience. When building inference for vertical AI-native apps, you want to add as many constraints as possible by getting specific with your use case.

1.2.1 AI-Native Applications

Generative AI models have unlocked a new class of applications across industries and domains. Each category of application relies on different models and modalities and requires tailored inference.

Category	Example	Considerations
Agents	Prospecting agent for sales teams	One user action triggers many inference calls
Chat	Front-line customer support chat with RAG	Time to first token makes chat feel fast
Voice	Real-time translation between languages	End-to-end latency for natural conversation
Media	Virtual try-on for clothes, shoes, and jewelry	Balance output quality vs. speed
Search	Legal document discovery	Offline corpus filling vs. online user requests

Category	Example	Considerations
RecSys	E-commerce product recommendations	Consistent latency with high request volume
Completion	Tab completion for coding in an IDE	Full completion chunk at user's typing speed
Moderation	Scan user-generated content for safety	High throughput for cost-effective checks

This is a tiny sample of the AI-native applications that are being built today, but it shows the breadth of considerations that inference engineers encounter. And as models get faster, cheaper, and smarter, new use cases that haven't even been imagined today will emerge.

1.2.2 Online versus Offline

One of the primary tradeoffs in inference engineering is latency versus throughput. Lower latency makes your application faster, but higher throughput makes it cheaper at scale because you can use fewer GPUs for the same number of users.

Most AI applications – code completion, chat, voice agents – are online applications that run in real time. With an impatient user waiting on the other end of every API call, these online applications should be optimized for latency.

However, some applications have offline batch inference needs. Offline jobs are better served by high-throughput model deployments where each individual request would be too slow for a good user experience, but the system as a whole processes far more requests per hour in parallel.

Some example offline workloads include:

- **Catalog transcription:** Transcribing a back catalog of podcasts, interviews, or other audio to make it searchable.
- **Document processing:** Embedding, converting, or analyzing a set of documents on a regular cadence.
- **Corpus preparation:** Cleaning, embedding, or otherwise preparing massive corpora of data for model training.

It's possible to have a single model that is used for both online and offline jobs. Whisper, a speech-to-text model, could be used in both a real-time dictation app and a batch transcription job. Assuming both use cases have enough volume, it will be more cost effective to create two separate deployments for the same model, with one optimized for latency and the other optimized for throughput.

1.2.3 Consumer versus B2B

Applications built for consumers and businesses have different inference needs.

Consumer applications are generally much more cost-sensitive and have less predictable usage patterns. Many consumer AI apps are designed for virality, and a single launch or marketing campaign can drive a spike in usage overnight.

Inference engineers working on consumer apps should prioritize marginal cost and flexibility while keeping latency and availability at a decent standard.

Business-to-business products often have better margins and more stable usage but require high availability and consistently low latency. Mission-critical software in the revenue path is held to a high standard of performance and reliability.

Inference engineers building for businesses must favor latency and uptime, though cost and scale are important secondary concerns.

In both consumer and business applications, compliance can limit infrastructure options, especially in regulated industries. Some essential considerations include:

- **Data sovereignty:** Are your GPUs in a geographic region where you are allowed to send user data?
- **User privacy:** Are the inputs and outputs of your model kept private and secure?
- **Regulatory compliance:** Are you and your underlying providers compliant with all relevant regulations?

Inference engineers must work closely with security and legal experts to ensure that the infrastructure they are operating is compliant.

1.3 Model Selection

All else being equal – hardware, runtime, optimizations, architecture – inference on a smaller model with fewer parameters will be faster and cheaper than inference on a larger model with more parameters.

That's why the most important decision in model performance optimization isn't the runtime engine or speculation algorithm, it's which model you choose to work with in the first place.

AI engineers iterating on early-stage products should just use prebuilt pay-per-token APIs for powerful frontier models like Kimi and DeepSeek (or even closed models like GPT and Gemini). Before product-market fit, it's not worth spending time or money on doing your own inference.

But when it's time to scale, the opposite advice applies. Find – or create – the smallest, easiest-to-run model that's smart enough to handle the task at hand. In many cases, this is still going to need to be a trillion-parameter frontier model. But it's always worth checking if a smaller, cheaper, faster model can get the job done.

Which model you pick also affects what inference optimizations are available to you. Inference engines vary in the depth of support for different model architectures. Stick with popular model architectures to ensure you'll find robust support across the performance tooling landscape.

1.3.1 Model Evaluation

Model evaluation, or evals, is the practice of systematically measuring model intelligence.

High conviction model evaluation is a prerequisite for inference engineering. Evals help inference engineers:

- **Spend time wisely:** Before investing in making a model fast, evals ensure the model is useful.

- **Establish a baseline:** Some performance optimization techniques risk reducing model quality, requiring a baseline to compare against.

Unlike standard intelligence benchmarks, which measure model capabilities against common tasks like MMLU or SWE-bench, evals are tailored to specific products, domains, and tasks.

Intelligence benchmarks are useful for shortlisting models, but they have become saturated or even gamed. Goodhart's Law states that "when a measure becomes a target, it ceases to be a good measure," and this applies to frontier labs' heavy incentive to show new world-record intelligence benchmarks with each model release.

While there are better ways to gauge overall model intelligence, like Elo rating on head-to-head win rate versus other models, there is no substitute for directly measuring how a model performs for your application.

A few tips for doing useful model evaluation work:

- **Look at your data:** Check eval results against your intuition for the product and problem space.
- **Be precise:** Have a clear idea of the hardest problems a model needs to solve and focus evaluation there.
- **Use tools:** Don't reinvent the wheel on one of the fundamental problems in AI engineering.

Appendix B includes recommendations for tooling and further reading about evals.

1.3.2 Fine-Tuning for Domain-Specific Quality

Fine-tuning is the practice of taking a pre-trained foundation model and adapting it to a specific use case by introducing new data.

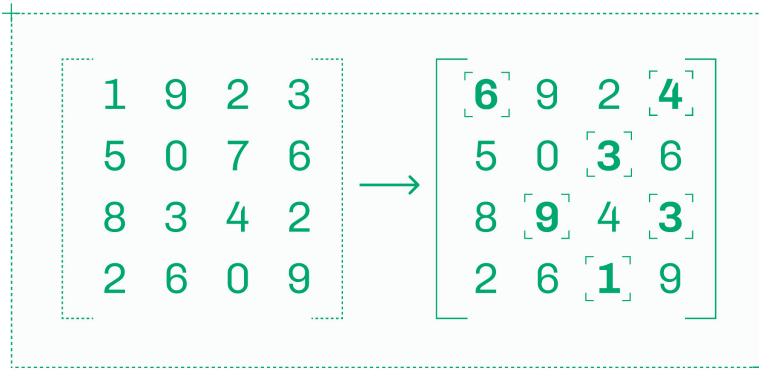


Figure 1.2: Fine-tuning a model changes the values of the model weights while keeping the same overall architecture.

If you can fine-tune a small model to pass your evals, you set yourself up for an easier time hitting your latency and cost targets for inference.

A great example of a domain where fine-tuning is effective is translating English into SQL, a language used to query databases.

General-purpose coding models are good at writing SQL, but these models are hundreds of billions of parameters. SQL is a relatively constrained language, so for an application that only needs to generate SQL queries from natural language prompts, a tiny fine-tuned model of just a few billion parameters can reach equivalent performance on this specific task.

Text-to-SQL is an extreme example – many domains won't support such a vast reduction in model size – that illustrates what's possible with a cleanly scoped domain, a strong set of evaluation criteria, and high-quality labeled data to use for fine-tuning.

1.3.3 Distillation

What if you could retain most of the intelligence of a large model at a fraction of the size? That's the idea behind distillation.

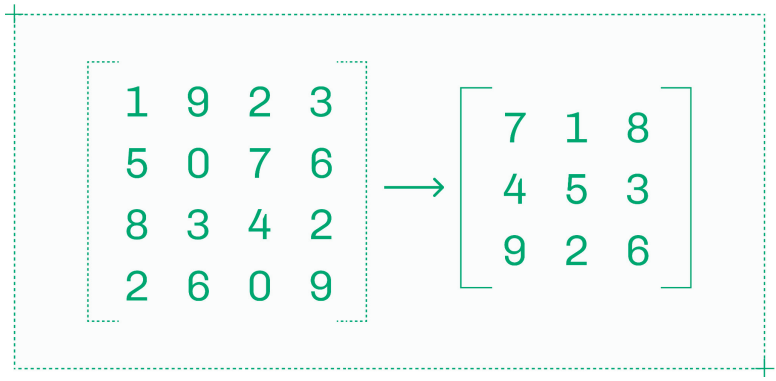


Figure 1.3: Distillation preserves the behavior of a large model in a smaller, more efficient model.

Distillation is the process of using a large "teacher" model to train a smaller "student" model to emulate the larger model's behavior. Unlike fine-tuning on synthetic data, where the model is trained on input-output pairs, distillation shows the student model the teacher model's actual probability distributions, not just its final answers.

Where fine-tuning teaches a model to perform better in a specific domain, distillation teaches the model how to emulate the behavior – good and bad – of a larger model.

Distillation sees substantially less real-world use than fine-tuning.

When a frontier lab releases a family of models of different sizes, the smaller models are generally not distilled from the larger ones. Instead, the models are independently trained to prevent the biases of the large models from artificially limiting the smaller models. But if the lab only trains a large model, distillation can make that model more accessible.

In January 2025, open model research lab DeepSeek released their then-flagship reasoning model DeepSeek-R1. As the model was so large (671B parameters), they also released distilled versions of the model on top of the most popular open model architectures at the time: Llama 3 and Qwen 2.5.

These distilled models showed similar reasoning behavior to the main DeepSeek-R1 model, albeit with worse intelligence benchmark scores, but the distilled models were relatively small and could take advantage of existing performance work for the Llama and Qwen architectures.

At publication, these DeepSeek-R1 distills are still among the most popular distilled models on Hugging Face, along with distills of models like Whisper for audio transcription and some image generation models.

1.4 Measuring Latency and Throughput

The two most common performance metrics for LLMs are TTFT (time to first token) and TPS (tokens per second). For modality-specific metrics beyond LLMs, see chapter 6.

Time to first token (TTFT)	**Tokens per second (TPS)**
With streaming output, how long does it take for a user to see the first output token?	How many tokens each second does the user receive after the first token is generated?
Based on compute-bound prefill	Based on bandwidth-bound decode
Lower TTFT == better latency	Higher TPS == better latency

While TTFT is a clear term, TPS is less precise. TPS can be a latency metric (tokens per second per user) or a throughput metric (tokens per second for the entire inference service).

Most people use TPS to mean a per-user latency metric. When needed, use more specific terms:

- **Perceived TPS:** The observed tokens per second per user after the first token (latency).
- **Total TPS:** The total number of tokens generated each second by the inference service (throughput).
- **Inter-token latency (ITL):** The time between subsequent tokens. An ITL of 10 milliseconds equates to 100 tokens per second per user.

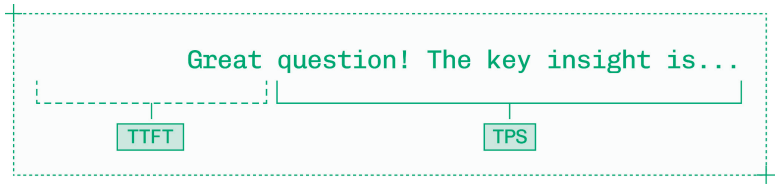


Figure 1.4: TTFT is the time it takes to generate the first token, while TPS measures how quickly subsequent tokens are generated.

TTFT and TPS are most commonly used for user-facing LLM systems like chatbots, where output is streamed to the user. For other requests, like a tool call for an agent, you instead measure latency as total response time as the tokens aren't useful individually.

1.4.1 Latency Percentiles

One important distinction when discussing and comparing metrics is what percentile you are measuring.

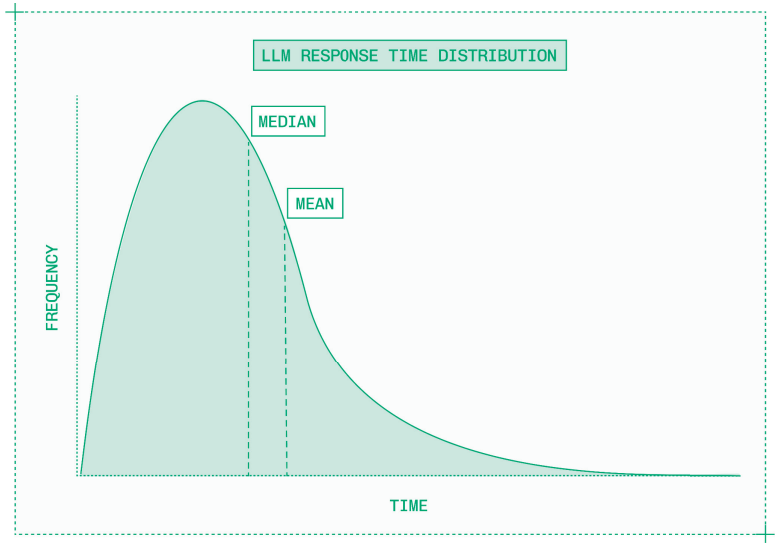


Figure 1.5: Mean latency is generally higher than P50 latency due to outliers in the right-skewed distribution of inference times.

The naive approach is to simply look at an average (mean) TTFT or TPS. However, this does not tell the whole story. LLM total response time is generally a right-skewed distribution, where most times concentrate around a mode, but outliers can take significantly longer.

These outliers can dramatically affect user experience and trust in a product. It's not good enough for most interactions to feel snappy if one in every ten takes several seconds.

Instead, inference engineers measure latency in percentiles.

P50	Median latency	1 in every 2 requests is slower
P90	90th percentile latency	1 in every 10 requests is slower
P95	95th percentile latency	1 in every 20 requests is slower
P99	99th percentile latency	1 in every 100 requests is slower

While driving down average latency matters, good performance work also focuses on reducing P90/P99 latencies for a more reliable user experience.

1.4.2 End-to-End Metrics

The other important distinction in metrics is whether you're measuring solely inference time – the on-GPU time required to generate tokens – or an end-to-end measurement that accounts for network latency and any queue time.

Both inference-only and end-to-end metrics are valuable to know. Inference time tells you how effective your model performance work is, while end-to-end metrics reveal your users' perception of how fast your application is. When inference time is fast but end-to-end time is slow, turn your attention to infrastructure rather than model performance optimization.

CHAPTER 2

Models

Models

Inference engineering is the practice of making generative AI models faster, less expensive, and more reliable – without sacrificing the quality that makes them so valuable. Both improving performance and preserving quality require a strong intuition for how models work under the hood.

Generative AI models are a composition of big, complex neural networks.

The history of neural networks stretches back to the 1950s, when the first perceptrons for simple binary classification were implemented in hardware. In the following decades, perceptrons were abandoned but then reinvented from single to multi-layer perceptrons with a new concept, back-propagation, which introduced hidden states between layers and a learning procedure that repeatedly adjusts weights within the network.

These neural networks had only a few layers. In the 2000s, research began into deep neural networks with dozens of layers. In 2012, AlexNet became the first deep neural network to show promising real-world capabilities and the effectiveness of GPUs for deep learning, leading to new architectures like word embedding models for text and Generative Adversarial Networks (GANs) for images.

But the story truly starts in 2017, when Vaswani and colleagues published the seminal paper "Attention Is All You Need," introducing the transformer. A transformer is a neural network with an attention mechanism that can learn relationships between various parts of a sequence.

Transformers are the foundation of generative AI. Transformers aren't just for LLMs, they power every modality of model from embedding to voice to image and video generation.

Across modalities, there are two important styles of transformer-based models:

- **Autoregressive token generation:** Start from a tokenized sequence and predict the most likely next token.
- **Iterative denoising:** Start from random noise and refine toward the most likely output via diffusion.

This chapter explores the architectural details of LLMs (autoregressive token generation) and image generation models (iterative denoising).

2.1 Neural Networks

Generations of research into neural networks form the theoretical foundation for generative AI.

To be a productive inference engineer, you need a basic intuition for essential concepts in neural networks. This section provides a high-level introduction; Appendix B offers recommendations for further reading.

The fundamental unit of a neural network is a node (a.k.a., neuron). A node is a short program that takes an input, multiplies it by some weights, adds some bias, and returns the result.

A group of nodes forms a layer. Nodes within a layer are independent of each other – they do their own calculations. The connection between nodes, or the "network" in a neural network, is between layers, where the nodes in a layer receive the output of the previous layer.

The neural networks behind LLMs contain dozens to hundreds of layers. There are three types of layers:

- **Input layer:** The first layer, which accepts and processes the input to the neural network.
- **Hidden layers:** Every layer between the first and last, which iteratively transform the input to arrive at an output.
- **Output layer:** The final layer, which returns the prediction from the network.

Each layer produces an output that the next layer reads as input. For the hidden layers, these outputs are called hidden states.

Hidden states are one kind of internal representation for data within a neural network. A key aspect of internal representation is its dimensionality, or the actual size of the vectors used.

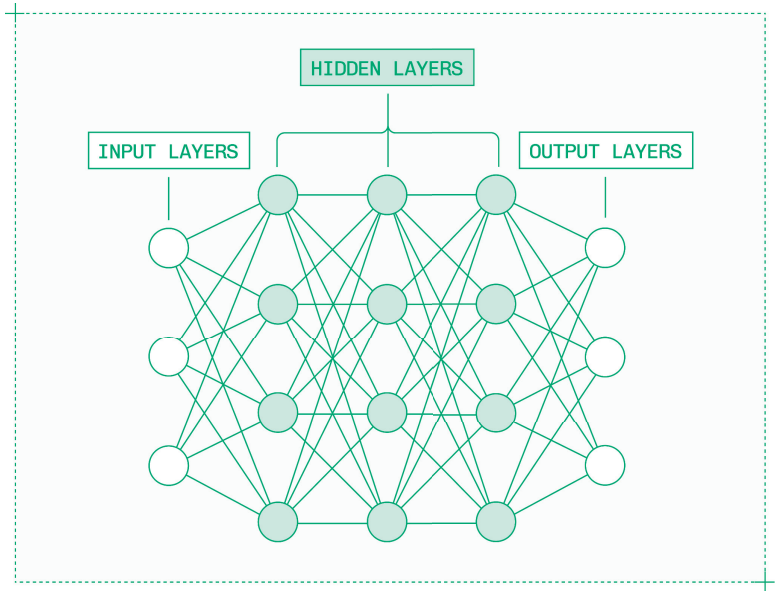


Figure 2.1: Multi-layer neural networks have one input layer, many hidden layers, and one output layer.

Internal representations for text input increase the dimensionality, encoding text chunks into vectors of hundreds or thousands of numbers to capture semantic meaning. But internal representations for image models reduce the dimensionality from millions of pixels down to a manageable size.

There are neural networks for creating these internal representations, and there are neural networks for using them:

- **Encoder:** Takes an input like text or an image and creates an internal representation of the input that includes additional information and semantic meaning.
- **Decoder:** Uses the internal representation to generate an output like text or an image.

Neural networks are composable. You can combine multiple neural networks together into a single model or use them sequentially to build a pipeline.

Modern LLMs are decoder-only, while encoder-only models are somewhat rare today, with old-school text embedding models from the BERT family as a prominent example.

Many models in other modalities use an encoder-decoder architecture. Whisper, a popular open model for audio transcription, uses an encoder to process audio input and a decoder to generate text tokens.

2.1.1 Linear Layers and Matmul

The most essential operation within a neural network is a matrix multiplication, or matmul. A matmul takes an input vector (a list of numbers) and a matrix (a grid of numbers) and multiplies the vector through the matrix to produce an output vector.

Within a neural network, a linear layer is the simplest form of matmul. Given an input vector, the linear layer applies a weight matrix and adds a bias vector:

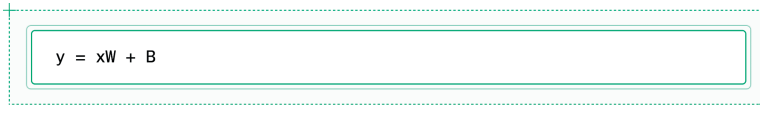


Figure 2.2: In a matmul, the output vector y is the product of an input vector x and a weights matrix W plus a bias vector b.

The weights of any given linear layer are a small part of a generative AI model's total weights, and the individual values within the weights matrix are set during training.

2.1.2 Activation Functions

Matrix multiplication is composable, meaning that multiplying a vector by two matrices is equivalent to multiplying that vector by the product of those matrices.

```
# Given the equation for two linear layers
y = xW1 + b1
z = yW2 + b2

# Multiply the weights matrices together
W3 = W1*W2

# And collapse the layers into a single operation
z = xW3 + b3
```

Figure 2.3: Two matmul equations representing separate layers collapse due to composition of linearity.

This is a problem for multi-layer neural networks because a series of linear layers, each one a matmul, would collapse into a single layer with all of the matrices multiplied together.

Deep multi-layer neural networks are useful because more layers use more parameters effectively and encode more meaning in hidden states.

Neural networks separate layers by breaking linearity with an activation function. Activation functions are non-linear to prevent composable matmul from collapsing layers, and are differentiable or mostly-differentiable to support back propagation.

One of the most basic activation functions in inference is ReLU, which stands for Rectified Linear Unit. ReLU is a simple function: if X is greater than zero, return X, else return zero. There are dozens of activation functions – including one named “Swish” thanks to its resemblance to the Nike logo – but most follow the same general pattern of mapping negative values to zero or near-zero, while keeping positive values unchanged.

Activation functions like ReLU, SiLU, Swish, and SwiGLU are fast to run, easy to train on (as they are mostly differentiable, they have a gradient at least for most values), and break linearity to support multi-layer neural networks.

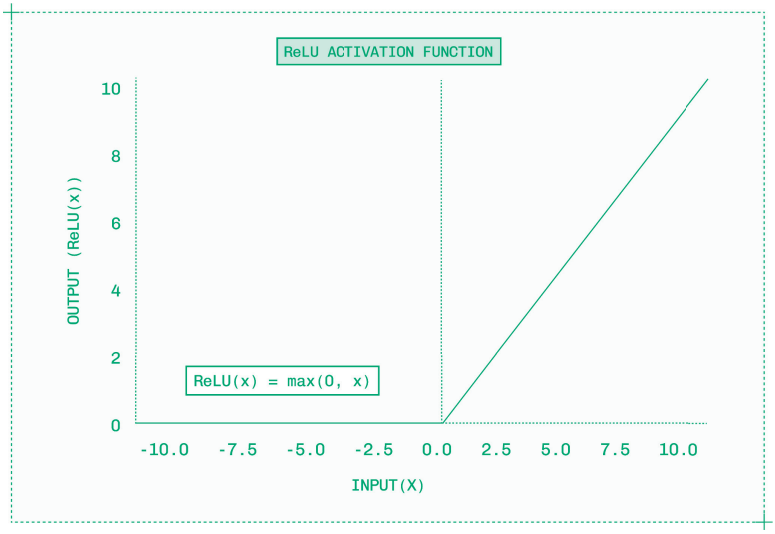


Figure 2.4: Activation functions like ReLU are used to break linearity in multi-layer neural networks.

2.2 LLM Inference Mechanics

LLMs are autoregressive token generation models. An LLM generates new tokens one at a time based on every previous token.

These tokens, the atomic units of language models, are numbers that represent chunks of text. Modern LLMs use subword tokenization, meaning that each token is a word or a fraction of a word.

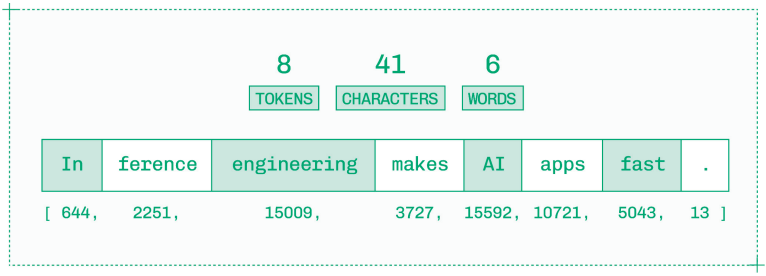


Figure 2.5: Subword tokenization uses one token per common word and punctuation mark, but it splits less common words into multiple tokens.

Converting text into tokens and tokens into text does not require any neural networks. Instead, a tokenizer is a simple mapping between strings and their numerical token representation.

A language model's vocabulary is the complete mapping between tokens and strings. Vocabularies and tokenizers vary from model to model, with more recent models employing more efficient tokenization schemes; the fewer tokens required to generate an output, the faster the end-to-end inference.

Most models have over 100,000 tokens in their vocabulary. In other modalities, like speech synthesis, the model vocabulary is expanded to let tokens represent other information like audio waveforms.

Inference involves two or three sequences of tokens:

- **Input sequence:** The prompt, chat, context, functions, and other input passed into the LLM.
- **Reasoning sequence:** Optionally, for reasoning models, an intermediate output sequence for thinking.
- **Output sequence:** The response generated by the LLM.

Combined, these sequences are limited to the model's context window (the total number of tokens the model can process and generate per request). A request may further limit the output sequence length with a `max_tokens` argument.

While the input sequence is a single string, LLMs are trained to accept varied inputs like multi-turn chat sequences with roles, function signatures for tool calls, and, in some cases, multimodal inputs. These inputs need to be combined into a single sequence. This is handled by the chat template, which differs subtly from model to model and must be implemented correctly in the inference engine.

Tokenizing the input sequence with the chat template applied is step zero for inference. Then, there are two primary phases of inference:

- **Prefill:** Process the input sequence to calculate attention for each input token and store those values in a KV cache.

- **Decode:** Perform forward passes through the model to generate tokens autoregressively.

Each forward pass in the decode phase must generate a token. This takes a few extra steps, as neural networks output vectors, not tokens.

The output layer of the neural network used in LLM decode generates a vector of logits. The length of this vector equals the model's vocabulary size. After normalization, these logits represent the probability of each potential token in the vocabulary being the correct output.

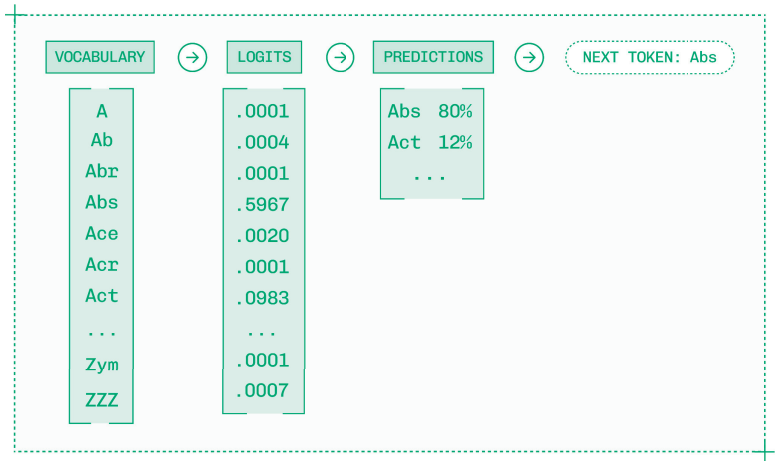


Figure 2.6: A decode pass generates a logit for each token in the model's vocabulary, then normalizes logits to percentages.

The output token is selected via a weighted random number generation based on the normalized probability vector. You can nudge that process via inference arguments:

- **Temperature:** Adjust the logits themselves before normalization.
- **Top-k:** Select the k most likely tokens after normalization, then re-normalize among them.
- **Top-p:** Select the smallest set of tokens after normalization whose probabilities add up to p.

A lower temperature, top-k, or top-p makes LLM output more predictable as the model is constrained to selecting highly likely tokens. Setting the temperature to 0 or top-k to 1 makes token selection deterministic (always select the highest-probability token).

When generating structured output, where the output conforms to a schema like JSON, there are additional tools like logit biasing for further directing the output of an LLM. These apply after each forward pass and support important LLM abilities like tool use; their correct implementation is essential for high-quality inference.

This logit generation and token selection process continues until the model decides that the stop token, a special value signifying the end of the output sequence, is generated (unless the context window or max tokens limit is hit first).

The two key pieces of this inference loop are generating the KV cache during prefill and generating output tokens (or more specifically the logit vectors that become tokens) during decode. These steps take the overwhelming majority of the time and resources during inference as they rely on large neural networks.

2.2.1 LLM Architecture

Every LLM on Hugging Face (the largest repository of open models) includes a `config.json` file: a few dozen lines detailing the model's architecture.

The architecture of a model is a collection of decisions made during the training process about the nature and shape of each component of the model. Within a single architecture, there may be:

- **Multiple sizes:** Models at different parameter counts, like Llama 8B and 70B.
- **Multiple variants:** The "base" and "instruct" variants of a given model share the same architecture.
- **Unlimited fine-tunes:** Methods like LoRA (Low-Rank Adaptation) change behavior, not architecture.

These architectures matter because they determine runtime and engine support. If you have a highly optimized deployment of a given architecture, you can deploy another variant of the same architecture and enjoy the same performance improvements.

Model architecture is one of the first lines in most configuration files. To parse an architecture name like `Qwen3MoeForCausalLM`:

- **Qwen:** The model family, or the brand name of the model.
- **3:** The major version of the architecture within the family.
- **MoE:** Indicates a Mixture of Experts model (section 2.2.4).
- **CausalLM**: Indicates a causal language model.

A causal language model predicts the next token in a sequence based on previous tokens, as opposed to, for example, a masked language model which fills in the blank based on surrounding tokens to the left and right. All generative LLMs today are causal language models.

Beyond the architecture's name, the `config.json` file contains information about the nature and dimensions of the various layers that form the underlying neural networks of a model and the vectors that pass through them during inference.

2.2.2 Transformer Blocks

The main body of an LLM is a series of dozens to hundreds of transformer blocks. These blocks form the core of a large neural network with three kinds of layers:

- **Embedding layer:** The input layer of the neural network takes tokens and returns embeddings.
- **Transformer blocks:** The hidden layers within the network are transformer blocks that generate a prediction.
- **Output layer:** Also known as a language modeling head or LMHead, converts the hidden states from the transformer blocks into a vector of logits, one for each token in the model's vocabulary.

Within the transformer blocks, there are sublayers for attention, a feed-forward neural network, and normalization.

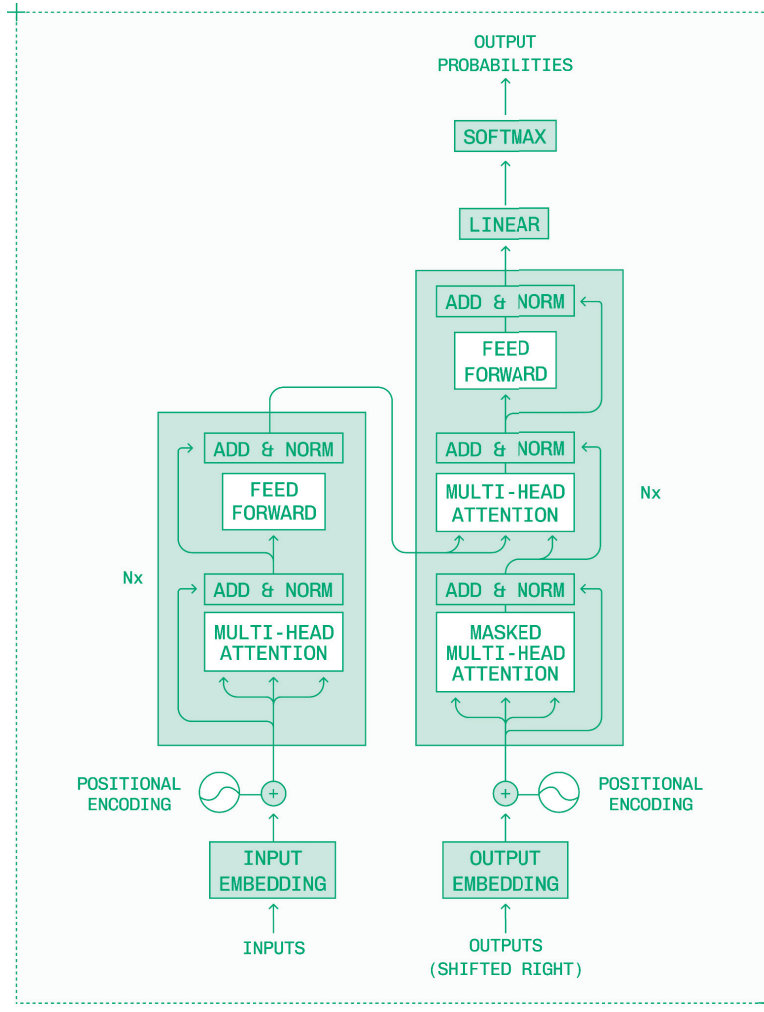


Figure 2.7: Transformer block diagram, adapted from "Attention Is All You Need" (Vaswani et al., 2017).

The feed-forward neural network is a multi-layer perceptron. These linear sublayers make up the majority of the trainable weights within an LLM, while the attention sublayers are the second-largest component. Other components like normalization and activation functions are a rounding error in the model's size.

While linear sublayers are the largest portion of the weights, the more complex operation for inference is attention.

2.2.3 Attention

Attention is the mechanism transformers use to relate a given token to other tokens in the sequence. Humans are good at interpreting the relationship between words. Attention brings the same capability to LLMs.

Consider the sentence "I decided to write a book because I thought it would be easy, but it was actually hard." Attention shows that the word "it" in the sentence refers to writing a book.

The standard form of attention is scaled dot-product attention, as shown in this equation.

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

Figure 2.8: The attention equation, adapted from "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness" (Dao et al., 2022).

Attention takes three inputs:

- **Q (queries):** The embedded representation of the token being generated or updated.
- **K (keys):** Representations of all prior tokens.
- **V (values):** Computed attention values for all prior tokens.

Attention sublayers within models are multi-head, where each head is one attention operation. If you visualize the architecture of a neural network, these heads are parallel to each other on the same sublayer. Each head could be responsible for attending to different kinds of relationships between tokens, like subject-verb agreement and co-reference resolution.

There are two main types of attention:

- **Self-attention:** Q, K, and V are all from the same sequence.
- **Cross-attention:** Q is from a different sequence than K and V, conditioning Q on external information.

LLMs use self-attention (with a causal mask to prevent looking ahead in the sequence), while image generation and multimodal models also use cross-attention (e.g., between the image being generated and the associated text prompt).

Because attention looks for relationships between the current token and every previous token in the sequence, it's a quadratic-time equation with respect to sequence length. As context expands, attention slows.

In practice, attention is linear, not quadratic, thanks to the KV cache. The KV cache is a standard component in attention implementation and stores key-value pairs for each previous token. By looking up this information in the KV cache rather than recomputing it each time, attention runs in linear time.

The KV cache is built during LLM prefill, used and updated during decode, and lives on GPU memory by default. Storing, accessing, and re-using the KV cache, a major topic in inference engineering, is covered in detail in section 5.3.

2.2.4 Mixture of Experts Models

The density of a neural network is determined by the number of connections between layers. Denser networks retain more information, while sparser networks take less compute and memory to run.

Mixture of Experts (MoE) is an architecture optimization that adds sparsity to linear layers. Rather than a single giant matrix, an MoE model has hundreds of smaller matrices (the experts) and routes each input to a small selection of experts. This is called activating the experts.

For an MoE model like Qwen3-235B-A22B, 22 billion parameters out of the model's 235 billion total parameters are activated per request. Thanks to

their low number of active parameters, MoE models are highly efficient for single-request local inference. However, in batched inference on production servers, different requests activate different experts, and you should expect almost all of the model parameters to be active at any given time unless sparsity is achieved in large-scale Expert Parallelism (section 5.4.2).

Expert routing is granular. Each forward pass through a model generates one token by working through every layer of the model. The router, a tiny model within the LLM, picks which experts to activate at each layer of the model. In the Qwen example, with 128 experts, the router picks eight experts at each of the 94 layers for every token that is generated.

MoE architectures are especially popular for larger models with 100B+ parameters, though there are MoE models as small as 20 to 30 billion parameters. Mixture of Experts unlocks a new form of inference parallelism called Expert Parallelism, which enables high-throughput inference for large models on multiple GPUs.

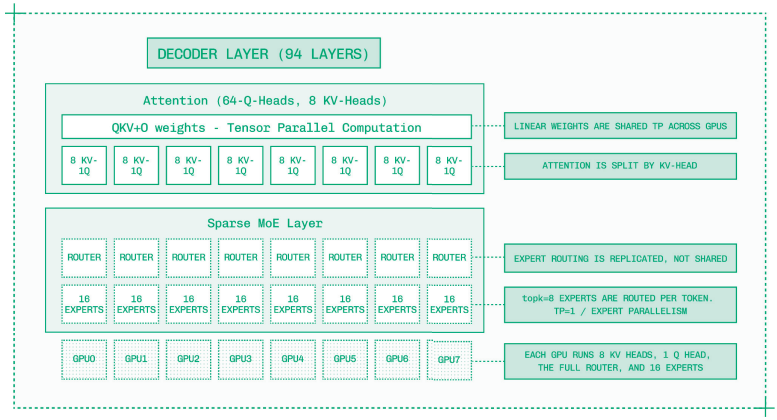


Figure 2.9: Mixture of Experts architecture includes both sharding and replicating to take advantage of multi-GPU inference.

Models under 32B parameters, and especially models under 8B parameters, tend to use traditional dense architectures efficiently. Domain-specific models for tasks like tab completion also don't gain much benefit from MoE as the entire model is effectively one expert.

2.3 Image Generation Inference Mechanics

Image generation models take a text prompt and create an image based on the prompt. These models slightly predate the public rise of LLMs, with both closed models from Midjourney and open Stable Diffusion models first released in the summer of 2022.

Image generation models aren't monolithic models like LLMs. Instead, they are pipelines of multiple models working together to generate images. Within a foundation model for image generation, there are three essential components:

- **Text encoder:** Converts the text prompt into instructions that the image generation model can understand.
- **Denoising model:** The heart of the model, iterates from noise to an image based on the prompt.
- **Variational Autoencoder (VAE):** Converts the model output from latent space to pixel space.

This pipeline mentality extends through the image model ecosystem. Beyond the base model, image inference often includes:

- **LoRAs:** Lightweight fine-tunes to change style and enhance quality.
- **ControlNets:** Outlines and edges to steer output images to match broad shapes and colors.

The vast and rich open-source ecosystem around image generation models includes tools like ComfyUI for building complex pipelines of foundation models and adaptions, swapping components to produce unique outputs.

The entire image generation pipeline operates in latent space. An ordinary HD image may be 1024x1024 pixels; that's well over a million pixels. As the denoising model needs to calculate attention over the entire image in parallel, it would be infeasible to work in pixel space.

Latent space is a low-dimensional representation of an image. A latent space matrix for an image may be 128x128, or about one percent of the total values of the pixel space it represents.

The latent space is initialized as random values, or noise. The denoising model refines that noise into an image over a series of steps based on the text prompt. Each step updates the entire latent space, unlike LLMs which process tokens one at a time. Most image generation models take 30 to 50 steps to create a high-quality image.

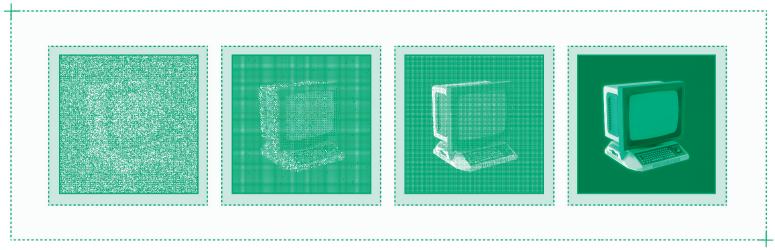


Figure 2.10: Diffusion-based models iteratively generate an image from noise, generally over 30 to 50 steps.

Within each step, the model runs two forward passes: one with conditioning (the text prompt) and one without conditioning. These generations are then combined based on a guidance scale. Because of this two-part step, a 50-step image generation actually takes 100 forward passes.

This process, and other essential parts of image generation, are controlled on a request-by-request basis via inference arguments. The most important arguments are:

- **Prompt:** Describes what the image should look like.
- **Negative prompt:** Separately describes any styles or objects that should not be in the image.
- **Number of steps:** Trades off speed and quality with the number of denoising steps, 30 to 50 for most models.
- **Guidance scale:** Controls the balance between creativity and prompt adherence, integer value generally around 4.
- **Image size:** Selects from a fixed menu of resolutions and aspect ratios for the output image.

While these core mechanisms are common across image generation models, their architecture has evolved considerably in the past few years.

2.3.1 Image Generation Model Architecture

Image generation models are built on transformers, specifically diffusion transformers. A diffusion transformer is very similar to the transformers that LLMs use, but instead of processing embedding representations of discrete tokens, it processes image data.

Diffusion transformers look at images in patches. When training a text-to-image model, the images in the training data are fed in via overlapping patches of 2x2 or 4x4 pixels, which are then embedded into latent space. Inference works in the opposite direction, with latent space transformed back to pixels once the image generation is final.

Image generation models are pipelines of multiple models, including a text encoder, denoising model, and variational autoencoder. A clean example of this pipeline is Stable Diffusion XL (SDXL). SDXL is an old model, but its architecture remains relevant.

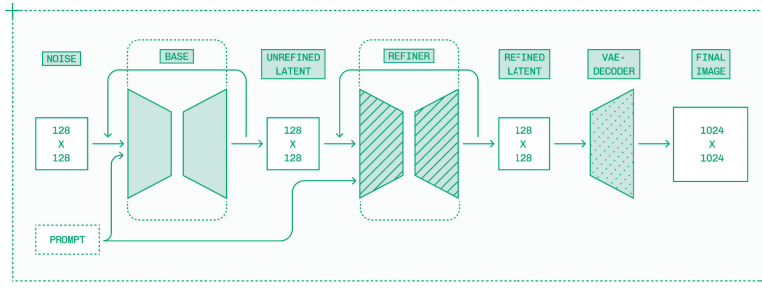


Figure 2.11: SDXL architecture pipeline, adapted from "SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis" (Podell et al., 2023).

SDXL's pipeline contains two diffusion models for denoising, the base and refiner. These models were trained for separate tasks: the base model goes from pure noise to a coherent image, while the refiner model adds details and ensures prompt adherence.

Modern models substantially outperform SDXL with better prompt adherence; accurate faces, hands, and details; legible text rendering; and support for image-to-image inference. These models, like the Qwen Image

family, are broadly similar to the SDXL pipeline, but with larger and more capable models at every step.

Component	SDXL (2023)	Qwen Image (2025)
Text encoder	CLIP-based model	Qwen 2.5 VL (7B)
Denoiser	<4B parameters	20B parameters
VAE	Single encoding	Dual encoding

The substantial growth in capability in modern image generation models like Qwen Image comes from larger component models and more complex pipelines. New abilities like legible text rendering and photorealistic human faces comes from switching from tiny NLP models to full LLMs for text encoding and increasing parameter counts on denoisers by a factor of five.

These larger models take more resources to run. Fortunately, as models have grown larger, GPUs have grown more powerful, though inference engineers can't rely on hardware gains alone to run these models efficiently.

The latest research direction in image generation models is blending diffusion transformer architecture with LLM architecture.

Anything that can be tokenized can be modeled as an LLM. LLMs have the advantage of baked-in text understanding, and they are already an important component of image generation pipelines.

LLMs solve many of the problems inherent to diffusion models. Where diffusion models can only produce a fixed size of output, LLMs are autoregressive and can produce a variable-length output. And where image models need up to 100 forward passes to generate an image, LLMs generate tokens in a single forward pass. Models like HunyuanImage-3.0 use this LLM-style architecture.

While this is a new frontier in image generation, there are other existing architectures for accelerating image creation and for extending image models to video generation.

2.3.2 Few-Step Image Generation Models

The most time-consuming part of image generation is the 30 to 50 denoising steps. Rather than making each step faster, what if there was a way to optimize image models by simply using fewer steps?

Few-step image generation models are trained to do just that: create high-resolution images with eight or fewer denoising steps. These models are 80 to 90 percent faster out of the box than traditional image generation models, though their output quality is noticeably lower.

There are two primary methods for creating these models:

- **Latent consistency:** Train a model to predict the target latent image vector directly and repeat the prediction two to four times to enhance quality.
- **Distillation:** Use adversarial distillation and/or progressive distillation to train a small model to emulate a larger one in fewer inference steps.

Distillation is more common than latent consistency today. When new image models like FLUX and Qwen Image are released, members of the open image model community create distillations in addition to quality and style-oriented LoRAs.

If you have a latency-sensitive use case where quality is less important, like real-time generative filters, consider few-step image generation models.

2.3.3 Video Generation

Video generation models are architecturally similar to image generation models, just bigger. They have three to five times more parameters and encode ten to one hundred times more information in latent space.

The naive approach to video generation is to work frame-by-frame. Early video generation used this framewise approach: first generating a starting frame, then using that frame to generate the next frame, and so forth.

The issue with a framewise approach is error accumulation. Small issues early on compound with each frame, and the video goes off the rails quickly.

Instead, modern video models hold the entire video in latent space and modify it on each denoising step. Each frame attends to each other frame and is updated on every forward pass.

If latent space for an image model represents two physical dimensions, X and Y, then latent space for a video model represents three dimensions: X, Y, and T (time).

The main limitation of this approach is that videos are a fixed number of frames, just like image generation models have fixed aspect ratios. Modern video generation models create sequences of a few seconds.

The constraint on video length is compute resources. Even with the latest GPUs, attention over massive latent space is extremely expensive, taking several seconds of inference for each second of video. Video generation models are so compute-intensive that they typically run with a batch size of one, meaning a full node of eight GPUs is working on a single request.

While attention on each denoising step is expensive, video generation models have the same total number of steps as image models, generally around 50 steps.

Video generation is a recent modality compared to LLMs and image generation. Their limitations line up with the limitations of LLMs two years ago.

LLMs (Late 2023)	**Video gen models (Late 2025)**
High TTFT, Low TPS	Slow generation times
Frequent hallucinations	Unrealistic physics
Maxed out Ampere GPUs	Max out Blackwell GPUs
Limited context windows	Short video outputs

Today, these limitations are mostly eliminated from LLMs. Removing them from cinematic video generation, as well as related areas like world models, 3D object generation (XYZ dimensions rather than XYT),

and other generative AI rendering models is a highly active area of research.

One important direction in research is coming back around to the idea of autoregressive generation, like with blending LLM architecture into image generation models. Rather than pure framewise video generation with its unusable error accumulation, new techniques like Self Forcing combine a global view of quality with an iterative approach to generation.

Adding autoregressive components to video models can partially address the bottleneck of attention, though it remains the most important and expensive component of inference.

2.4 Calculating Inference Bottlenecks

In a perfectly optimized system, every resource is fully utilized at all times. In GPUs, there are two main resources:

- **Compute:** The number of floating-point operations per second that the GPU can achieve.
- **Memory bandwidth:** The number of bytes that the GPU can move per second.

Ideally, compute is never sitting idle waiting for information from memory, and memory bandwidth never goes unused waiting for compute to finish.

In the real world, systems have bottlenecks: imbalances where one resource is idle while another is saturated. Discovering these bottlenecks is the first step to improving performance. If a certain operation is bottlenecked on memory bandwidth, no amount of compute optimization will make the system faster, and vice versa.

In most cases, inference systems have the following bottlenecks:

- LLM prefill (KV cache construction) is compute bound.
- LLM decode (token generation) is memory bound.
- Image and video generation are compute bound.

When optimizing performance on each of these phases, the goal is to make the bottleneck less limiting to system-wide performance. For example, batching multiple requests together makes LLM decode less memory bound because processing a batch of requests uses more compute for the same amount of memory traffic.

2.4.1 Ops:Byte Ratio and Arithmetic Intensity

Each GPU has a specific compute speed (measured in operations per second) and memory bandwidth (measured in gigabytes or terabytes per second). Compare these to determine the ops:byte ratio of a given GPU.

For example, an H100 GPU in FP16 can perform 989 teraFLOPS of dense computation against 3.35 TB/s of memory bandwidth. This yields an ops:byte ratio of about 295.

For inference in FP16 to be perfectly balanced (as all things should be) on an H100 GPU, the inference system needs to perform 295 floating point operations for every byte of memory it accesses.

To figure out that ratio, calculate the arithmetic intensity of the algorithm. Arithmetic intensity, also known as operational intensity, is the ratio between work and memory traffic for the calculation at hand.

```
intensity = work / memory_traffic
```

Figure 2.12: The equation for arithmetic intensity.

Where ops:byte was measured on a per-second scale, arithmetic intensity is measured across the execution of a single function or algorithm.

Arithmetic intensity is visualized with a roofline model, which charts performance against the bandwidth ceiling (a diagonal line) and the performance ceiling (a horizontal line).

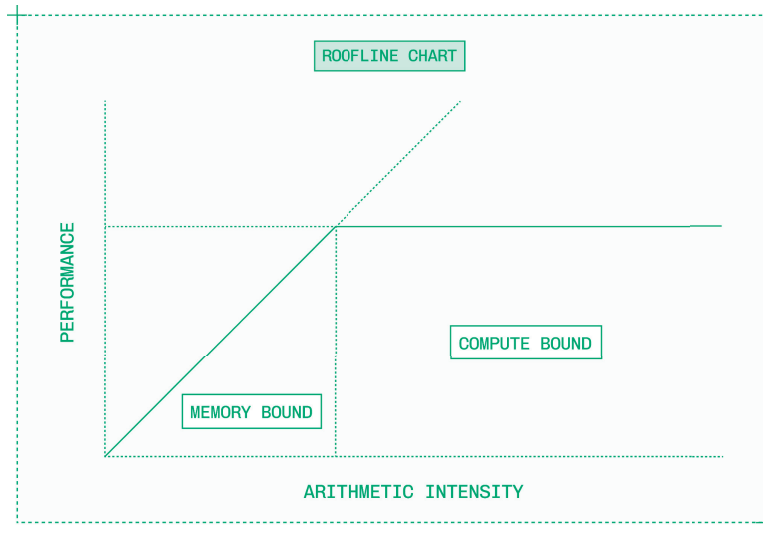


Figure 2.13: A roofline chart shows the switch from memory to compute bottleneck based on arithmetic intensity.

Plotting against the roofline model reveals if the algorithm is:

- **Compute bound:** When the arithmetic intensity is higher than the hardware's ops:byte ratio and hits the horizontal performance ceiling, it's compute bound.
- **Memory bound:** When the arithmetic intensity is lower than the hardware's ops:byte ratio and hits the diagonal bandwidth ceiling, it's memory bound.

To find a bottleneck, look at arithmetic intensity for the most expensive calculations in a system. For inference, one such calculation is attention.

2.4.2 LLM Inference Bottlenecks

LLM inference has two phases:

- **Prefill:** Determines the time to first token (TTFT) and is compute-bound.
- **Decode:** Determines the tokens per second (TPS) and is memory-bound.

For each phase, you can prove the existence of the bottleneck by comparing the arithmetic intensity of the most important operation to the ops:byte ratio of available hardware.

In both prefill and decode, the most expensive operation is attention. The exact arithmetic intensity of attention depends on the model architecture (dimensions, heads, etc), the input sequence length, and the implementation of the attention algorithm.

The essential difference is that prefill processes the entire input sequence in parallel, while decode generates tokens one at a time.

For prefill, the model weights are loaded a single time, then a series of large matrix multiplication between the matrix of inputs and the attention matrices occurs. This is a lot of calculations versus a single read from memory, creating a high arithmetic intensity.

On decode, the model weights are loaded for every token, which is generated via much less-expensive vector-matrix multiplication. In this case, relatively few floating-point operations are needed compared to loading the entire model weights, so the arithmetic intensity is low.

As an example of calculating exact arithmetic intensity, consider a decode step for a model with a 128-dimensional attention head (d=128) on a sequence of 4096 tokens (N=4096). For this analysis, use the standard algorithm for attention without any optimizations.

ALGORITHM O: Standard Attention Implementation

REQUIRE: Matrices **Q**, **K**, **V** $\in \mathbb{R}^{Nxd}$ in HBM.

1. Load **Q**, **K** by blocks from HBM, compute **S** = **QK**^{T}, write **S** to HBM.
2. Read **S** from HBM, compute P = softmax(**S**), write **P** to HBM.
3. Load **P** and **V** by blocks from HBM, compute **O** = **PV**, write **O** to HBM.
4. Return **O**.

Figure 2.14: Standard attention implementation, adapted from “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness” (Dao et al., 2022).

Based on the parameters of this exercise, establish the size of these matrices:

- **N:** The sequence length, established as 4096.
- **d:** The dimensionality of the attention head, set to 128.
- **Q, K, V:** Given as Nxd, or 4096x128.
- **S, P:** Calculated as NxN, or 4096x4096.
- **O:** Calculated as Nxd, or 4096x128

Assume FP16 inference, where each value in the matrix is two bytes. For reference, a 4096x4096 matrix is about 32 MiB, or about the same amount of data as a high-resolution RAW DSLR photo.

Each of the three lines of the attention algorithm follows the same pattern: load data from memory, perform a calculation, and store the result to memory.

READ	COMPUTE	WRITE
Q=2*N*d K=2*N*d	S=(2*d)*(N*N)	S=2*N*N
S=2*N*N	P=3*(N*N)	P=2*N*N
P=2*N*N V=2*N*d	O=(2*N)*(N*d)	O=2*N*d

Figure 2.15: Memory movement (reads and writes) and compute work for the attention implementation in Figure 2.14.

To calculate the total memory movement, sum the first and third columns, which track reads from and writes to GPU memory:

```
# Total memory movement in bytes

= (2*2*(N*d))+(2*(N*N))        +
  (2*((N*N)+(N*d)))+(2*(N*N)) +
  (2*(N*N))+(2*(N*d))

= 8N² + 8Nd bytes
```

Figure 2.16: The total memory movement for a kernel is the sum of all reads and writes across the three steps.

To calculate the total compute, sum the second column:

```
# Total compute in floating point operations

= ((2*d)*(N*N)) +
  (3*(N*N))     +
  ((2*N)*(N*d))

= 4(N²)d + 3N² ops
```

Figure 2.17: The total compute for the kernel is the sum of operations across the three steps.

To calculate the arithmetic intensity, compare the work (total compute) to the memory traffic:

```
# Arithmetic intensity

= 4(N²)d + 3N² ops / 8N² + 8Nd bytes

= 62 ops:byte
```

Figure 2.18: The arithmetic intensity of a kernel is the total work (number of compute operations) divided by the memory movement

For this example, the arithmetic intensity of 62 is much lower than the H100 GPU's ops:byte ratio of 295. The exact numbers vary by model, sequence length, and hardware, but this example illustrates the general principle that decode is memory bound.

Calculating arithmetic intensity like this is an academic exercise, not a routine task for inference engineers. But seeing it once is useful for building intuition.

2.4.3 Image Generation Inference Bottlenecks

Image and video models are relatively small – they have a tenth as many parameters as frontier language models – but their attention mechanism is just as computationally demanding.

Image and video generation models use iterative denoising, not autoregressive token generation.

Just like attention for LLM prefill processes the entire input sequence at once, attention for generating media must consider the entire image or video object as represented in latent space.

Also like LLM prefill, image and video generation model inference is bottlenecked on compute. Specific techniques for optimizing inference for these modalities are featured in sections 6.5 and 6.6.

2.5 Optimizing Attention

For LLMs, attention scales quadratically with the length of the input sequence. Each calculation of attention depends on the K and V values of each previous token. In practice, attention is a linear-time operation during decode as the KV cache stores the results of key and value computations for previous tokens.

Even a linearly scaling algorithm gets very expensive. Attention is one of the most expensive parts of inference across models and architectures. Naturally, optimizing attention is an important and highly active research area.

Attention is a sensitive process because each token depends on every previous token. Small errors in attention can accumulate quickly, making attention optimization a delicate process.

Figure 2.14 showed that the attention algorithm itself is straightforward. However, that basic implementation is inefficient. The intermediate matrices `S` and `P` are stored at the end of one step, then immediately loaded in the next step.

There are two strategies for optimizing attention:

- **Implementation improvements:** Write higher-performance kernels that use memory and compute more efficiently.
- **New algorithms:** Create algorithms for attention that scale in better-than-quadratic time with minimal quality loss.

Implementation improvements are still limited by attention's quadratic time complexity, but are lossless (do not affect quality) and make inference feasible for long sequences on today's hardware. Other algorithmic approaches trade off quality for time and space complexity, though training techniques can minimize the impact.

The most famous implementation of attention is the FlashAttention series of papers and kernels. Where the basic algorithm can be implemented in a handful of lines of code, FlashAttention uses tens of thousands of lines to implement attention in hand-fused kernels built for specific GPUs – FlashAttention for H100 uses different code than FlashAttention for B200.

FlashAttention works by eliminating excess reads and writes from memory and laying out the attention algorithm to precisely fit the GPU's capabilities. FlashAttention is especially useful for compute-bound operations like LLM prefill and video generation.

Another important implementation is PagedAttention. KV caches quickly grow large, filling GPU memory and taking time to read. PagedAttention partitions the KV cache into blocks (pages) that can be accessed via a lookup table. This means the KV cache can be stored across the GPU with fragmented memory rather than requiring a single contiguous block of memory.

While FlashAttention and PagedAttention are valuable optimizations, they don't change the fact that attention is a quadratic algorithm. New variants of attention improve the underlying time and space complexity:

- **Sliding window attention:** Computes attention for a sliding window of the previous `w` tokens, turning attention from `O(N^2)` to `O(Nw)` where `w` is often in the range of 8K to 32K.

- **Gated attention:** Various types of layers introduced in training allow for approximating attention for certain chunks of context in linear time with respect to chunk length.
- **Linear attention:** Replaces the quadratic softmax equation with a linear-time algorithm that approximates attention.
- **Compressed attention:** Periodically compresses context from earlier in the sequence, attention considers both compressed context and uncompressed recent tokens.
- **Multi-latent attention:** Approximates attention in low-dimensional latent space.

Intuitively, it makes sense that tokens near each other in a sequence affect each other more than tokens from much earlier. The sentence I am writing now follows closely from the previous sentence, but less so from the sentence at the start of this chapter.

This intuition can be extended through training. Algorithms like sliding window attention, when applied during training, create models that keep quality high when the same technique is used in inference.

Another avenue of research is avoiding attention altogether by using a different architecture than transformers. Mamba is a selective state-space model that replaces self-attention with a recurrent state update, achieving linear scaling on sequence length. Hybrid models sometimes mix Mamba-style state-space model blocks with transformer blocks. Applications of state-space models are still limited, though hybrid models are becoming more popular with open models like NVIDIA Nemotron 3 Nano adopting hybrid architectures.

CHAPTER 3

Hardware

Hardware

Inference engineering relies on accelerators: powerful hardware designed to load terabytes of data and perform trillions of operations per second.

The most common type of accelerator for inference is the GPU, and the market leader in GPUs for inference is NVIDIA. This book focuses on inference engineering for NVIDIA GPUs in the datacenter, but section 3.4 of this chapter covers other vendors of datacenter accelerators, and section 3.5 covers local inference.

Across vendors, there are three types of GPUs on the market:

- **Datacenter GPUs:** Racked servers with interconnected high-performance GPUs. Example: NVIDIA B200.
- **Workstation GPUs:** Individual desktop GPUs for professional workflows. Example: NVIDIA RTX Pro 6000.
- **Personal computing GPUs:** Individual desktop GPUs for everyday use. Example: NVIDIA GeForce RTX 5090.

Inference at scale uses datacenter GPUs mounted on racks: refrigerator-sized chassis with standardized power, networking, and cooling.

Datacenter GPUs like the NVIDIA B200 offer the highest individual performance, but more importantly, include high-bandwidth GPU-to-GPU interconnects, are installed in highly standardized configurations, and are available by the millions in datacenters worldwide.

I doubt that you have a B200 GPU running under your desk. If you do, send me a picture! Instead, inference on datacenter GPUs runs in one of three modes:

- **Cloud:** GPUs are rented in someone else's datacenter, usually hyperscalers like AWS and GCP or neoclouds like Coreweave and Nebius.
- **On-premise:** GPUs are purchased and installed in a datacenter that you control directly.
- **Air-gapped:** GPUs are installed on-premise and you need to physically access the GPUs to run inference.

Most inference engineers use cloud GPUs. Large enterprises and governments run on-premise and air-gapped deployments, but cloud-based GPUs offer the flexibility and access that fast-growing AI products need to scale.

Even with these constraints, navigating the hardware landscape is complex. From variations among cloud providers to NVIDIA's own naming conventions, there are many nuances in selecting the right accelerator.

3.1 GPU Architecture

GPUs are throughput machines. Where CPUs are great at complex sequential execution, GPUs are designed for simple, massively parallel workloads.

Specifically, GPUs are great at performing one uniform operation on thousands of independent pieces of data. Given that AI model inference is a series of vector and matrix multiplications, GPUs are a natural fit.

While the principle of highly parallel computation is simple, GPUs themselves are extraordinarily complex pieces of technology. Hardware engineering is a fascinating multidisciplinary field, from the physics of powering and cooling the chips to the impossibly tight tolerances involved in manufacturing each component.

Inference engineers work at a comfortable level of abstraction above the GPU hardware, but a strong mental model for what's going on inside the box is essential for building high-performance systems.

3.1.1 Compute

If you're familiar with CPUs, you've probably heard about cores, like an 8-core Intel i9 CPU in a high-end gaming computer.

In GPUs, cores have a different meaning. GPUs have Streaming Multiprocessors (SMs), while each SM contains multiple cores. There are three types of compute in GPUs:

- **CUDA Core:** Operates on individual numbers (scalars).

- **Tensor Core:** Operates on vectors and matrices.
- **Special Function Unit (SFU):** Accelerates certain mathematical operations like sin, cos, and log.

When measuring GPU compute for inference, measure in terms of Tensor Core compute. SFUs are essential for softmax, but Tensor Cores are responsible for Matrix Multiply and Accumulate (MMA) instructions, which are foundational to inference.

The "accumulate" step in MMA means adding the product of two matrices to a base matrix to produce the output, as shown in Figure 3.1.



Figure 3.1: Matrix Multiply and Accumulate (MMA) multiples matrix A by matrix B, adds matrix C, and stores the result as matrix D.

Unlike cores, the concept of a thread is similar between CPUs and GPUs. Where a CPU has dozens to hundreds of threads, GPUs have tens to hundreds of thousands of threads that can work concurrently, switch tasks in a single clock cycle, and execute simple instructions in parallel.

Compute is measured in FLOPS (floating point operations per second) and datacenter GPUs are capable of trillions or quadrillions of FLOPS (teraFLOPS and petaFLOPS, respectively). However, when you read a spec sheet, you'll see two measurements for Tensor Core compute:

- **Dense:** The raw floating-point operations per second if every element of the tensor is used.
- **Sparse:** In tensors with 2:4 structured sparsity, where 50 percent of the values are 0, Tensor Cores can skip multiplication by 0.

A GPU's FLOPS at a given precision with sparsity are often, but not always, double that of dense operations at the same precision. By default, inference is dense, so ensure that you're looking at FLOPS without sparsity.

FLOPS generally double with each halving of precision. A GPU capable of one petaFLOPS on 16-bit numbers will be able to do two petaFLOPS on 8-bit numbers. This is relevant for inference – be sure to compare FLOPS across GPUs at identical precisions.

Compute is the bottleneck for LLM prefill and for image and video generation. If you're selecting hardware with one of these use cases in mind, pick the accelerator with more FLOPS.

3.1.2 Memory and Caches

GPUs contain high-speed onboard memory called VRAM. Just like the "G" in GPU stands for graphics, the "V" in VRAM stands for video – a callback to these accelerators' original purpose.

Today, VRAM is added to GPUs in the form of HBM3, HBM3e, or HBM4, all various generations of high-bandwidth memory. GPUs have dozens or hundreds of gigabytes of VRAM.

There are two types of memory on any chip, CPU or GPU:

- **DRAM (Dynamic RAM):** General-purpose off-chip memory denominated in gigabytes.
- **SRAM (Static RAM):** Faster, more expensive, on-chip memory denominated in kilobytes or megabytes.

VRAM is a type of DRAM. GPUs also feature SRAM on-chip in the form of caches. GPUs have three levels of cache:

- **L0:** Instruction cache for a single Tensor Core.
- **L1:** Shared memory per Streaming Multiprocessor.
- **L2:** Global cache across Streaming Multiprocessors.

An H100 GPU has 256 KB of L1 cache per Streaming Multiprocessor and 50 MB total L2 cache on chip.

VRAM bandwidth measures the peak transfer rate between GPU cores and VRAM via the memory bus. In practice, this determines how quickly VRAM can supply data into the GPU's cache hierarchy.

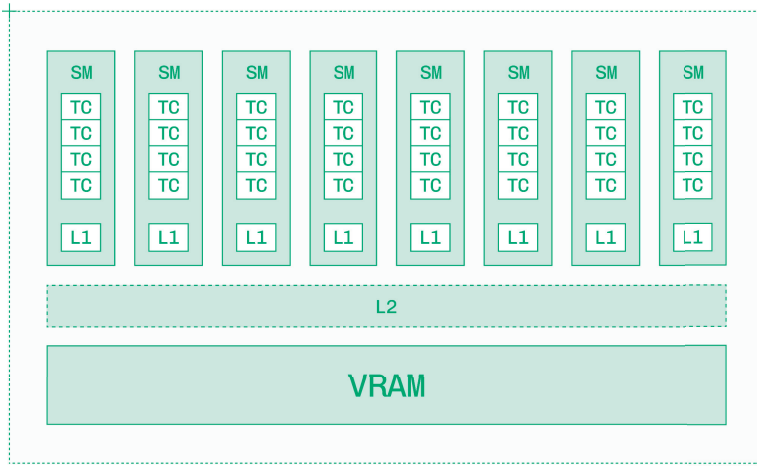


Figure 3.2: A GPU has multiple SMs, each of which has multiple Tensor Cores. L1 cache sits within SMs and L2 cache is shared.

The total amount of VRAM on a GPU limits the size of the model you can load onto it. The VRAM should hold the model weights, plus at least 50 percent headroom for KV cache (more for long context, high batch sizes, or video generation models).

If there isn't enough VRAM available for the weights, loading the model will fail with an OOM (out of memory) error. And if there isn't enough headroom, inference will be slow or crash with an OOM.

Memory bandwidth is the bottleneck for LLM decode at low to medium batch sizes. High-end GPUs have terabytes per second of memory bandwidth. If you are picking a GPU and want to generate more tokens per second, select the accelerator with a higher memory bandwidth, like the H200 instead of the H100.

3.2 GPU Architecture Generations

Hardware iteration cycles are slow. There are years of lead time between finalizing the architecture design and shipping GPUs.

Given the speed of the AI industry, this tapeout and testing process means that even next-generation GPUs were designed at a time when AI model

capabilities looked nothing like they do today. Designing a GPU architecture with staying power on the market requires foresight into how use cases will evolve over the expected lifetime of the GPU.

For years, training AI models was the primary use for GPUs. Now, with inference rising as the dominant use case, the latest architectures are introducing inference-focused features.

GPU names, like B200, have two parts:

- **Letter:** Signifies the architecture generation used in the chip.
- **Number:** Identifies individual models within the generation.

For example, the H100 directly replaces the previous generation A100, while the H200 is a larger GPU within the same generation (and is in turn supplanted by the B200).

The numbering appears somewhat arbitrary, with different sets of numbers used from generation to generation, but the general rule is that a bigger number means a larger, more powerful, more expensive GPU. On the other hand, the lettering for architecture is very meaningful.

Every one to two years, NVIDIA releases a new GPU architecture, which powers all of their products from the datacenter to personal computers. Each architecture generation introduces both improved base speeds on compute and memory and new features for more efficient inference.

Since 1998, NVIDIA has named their GPU architectures for prominent scientists.

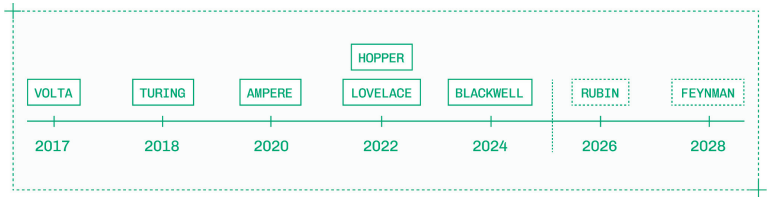


Figure 3.1: NVIDIA GPU architecture names from 2017 through all announced future architectures.

While GPU architectures go back decades, inference engineers generally work within the three to five most recent generations of GPUs. Even for cost-sensitive workloads, modern architectures' efficiency often makes them more cost-effective for large scale traffic, and of course newer architectures offer better performance.

You may still see Turing (T4) and Ampere (A10, A100) GPUs from time to time in low-traffic or legacy systems, but most deployments today use Lovelace (L4, L40), Hopper (H100, H200), or Blackwell (B200, B300) GPUs.

The Hopper and Blackwell architectures offer low-precision Tensor Cores, high-bandwidth memory, and inference-focused features, while Lovelace GPUs are used for low-cost inference on small models. The upcoming Rubin and Feynman architectures promise even greater performance when released in 2026 and 2028, respectively.

3.2.1 Hopper GPUs

GPU	FP8 compute (dense)	Memory	Bandwidth
H100	1,979 teraFLOPS	80 GB	3.35 TB/s
H200	1,979 teraFLOPS	141 GB	4.8 TB/s

The Hopper architecture, named for Rear Admiral Grace Hopper, was first released in March 2022 with the H100 GPU.

Hopper introduces support for FP8, a floating-point number format with 8 bits of precision. FP8 Tensor Cores are twice as fast as FP16 Tensor Cores, and moving FP8 values around takes half as much memory bandwidth. As section 5.1 discusses, this does not linearly translate to double the performance, but for workloads that can be run in FP8, it's a significant gain.

The Hopper architecture adds fourth-generation Tensor Cores within more and faster Streaming Multiprocessors than previous generations. Alongside dynamic programming instructions, thread block clusters, and distributed shared memory, Hopper GPUs give CUDA engineers more tools for writing high-performance kernels.

One such kernel is FlashAttention 3, which improves performance and memory efficiency for attention on Hopper GPUs. FlashAttention 3 takes advantage of the new asynchronous data transfer and execution features introduced with Hopper.

The H100 and H200 GPUs are among the most widely used inference accelerators, and for good reason. The Hopper architecture is new enough to be performant but established enough for industry-wide support and highly optimized kernels, and H100 and H200 GPUs are right-sized for common workloads across all modalities.

3.2.2 Ada Lovelace GPUs

GPU	FP8 compute (dense)	Memory	Bandwidth
L4	242 teraFLOPS	24 GB	300 GB/s
L40	362 teraFLOPS	48 GB	864 GB/s

The Ada Lovelace architecture, named for the first computer programmer, was first released just six months after Hopper.

The two architectures are similar, with Lovelace acting as more of a counterpart than a successor. Lovelace also supports FP8 inference.

Where Hopper is focused on AI applications, Lovelace GPUs are more graphics oriented. Lovelace GPUs do not support NVLink interconnect. This is a major limitation. Nodes with eight Hopper or Blackwell GPUs use these high-bandwidth interconnects for efficient parallelism. Lovelace GPUs must be used individually or via inefficient parallelism methods like Pipeline Parallelism.

L4 GPUs can be a cheap and convenient way to run small models for modalities like text embeddings and computer vision.

But L40 GPUs generally aren't a great choice for inference. For the same memory footprint, multi-instance GPUs (section 3.3.2) offer much higher compute and memory bandwidth on fractional H100s.

3.2.3 Blackwell GPUs

GPU	FP8 compute (dense)	Memory	Bandwidth
B200	~5 petaFLOPS	192 GB	Up to 8 TB/s
B300	~5 petaFLOPS	288 GB	Up to 8 TB/s

The Blackwell architecture, named for mathematician David Blackwell, was first released in November 2024 with the B200 GPU, followed by the B300. While the B100 does exist, it's not common for inference.

Where Hopper introduced FP8, Blackwell goes further in low-precision computing with FP4, a 4-bit floating point format, plus a set of microscaling formats (MXFP8, MXFP4, NVFP4) for better quality retention during inference. Section 5.1 explains these formats in detail.

Blackwell builds on Hopper's asynchronous programming paradigm with more features for loading and storing between tensor and global memory. The updated FlashAttention 4 kernel relies heavily on tiling loads, computations, and writes in asynchronous pipelines.

The B200 and B300 are the new gold standard for inference, offering the highest performance for large language models and demanding workloads like video generation. Software support, optimized kernels, and general availability for Blackwell GPUs have all come online in recent months, marking an important transition in the inference industry.

3.2.4 Rubin GPUs

The Rubin architecture, named for astronomer Vera Rubin, will launch in 2026 as NVIDIA's next-generation GPU architecture.

When evaluating new architectures, it's important to reserve judgement until you can run real-world performance benchmarks. New architecture rollouts, followed by industry-wide software support, take a year or so to fully saturate.

At the time of publication, there are some concrete details about Rubin. The Rubin architecture uses HBM4, an upgrade from the HBM3 and HBM3e that has powered the last few generations of GPUs. Inference

tasks like LLM decode that are bound on memory bandwidth will benefit from this higher-throughput VRAM.

Rubin also introduces the new CPX, a separate chip that's built for compute-bound tasks like LLM prefill. The CPX will be part of NVIDIA's rack-scale systems for high-volume inference.

After Rubin, NVIDIA will release Feynman in 2028. Few details are known about Feynman, but it is likely to support larger and more powerful chips with a faster memory architecture.

3.2.5 Grace and Vera CPUs

NVIDIA also offers its own ARM-based CPUs, which are integrated with their GPUs on superchips like the GH200 and GB200. The "G" stands for "Grace," as in Grace Hopper (to match Hopper GPUs).

NVIDIA Grace GPUs have overall strong compute performance, but what matters for inference is that they have a much higher-bandwidth connection between the CPU and GPU. Grace CPUs use NVIDIA NVLink Chip to Chip for up to 900 GB/s bi-directional bandwidth between the CPU and GPU memory.

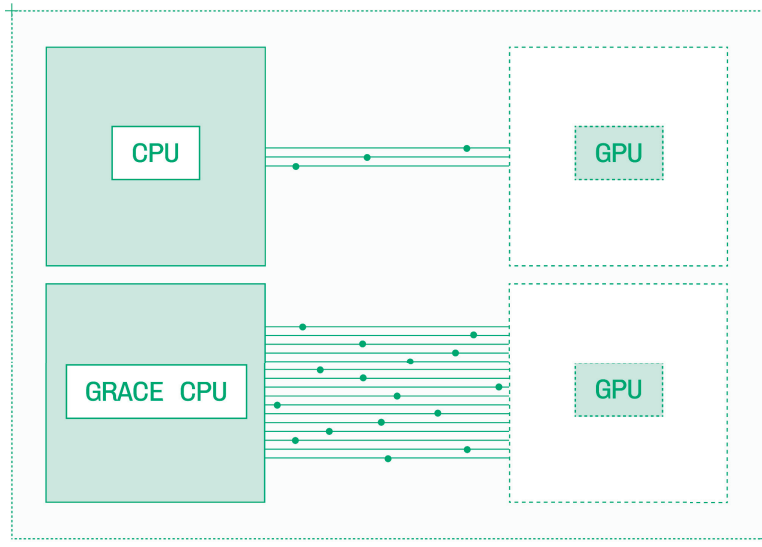


Figure 3.3: Grace CPUs have a higher-bandwidth interconnect to the GPU than standard CPUs do.

This CPU-to-GPU interconnect is several times faster than PCIe or other standard connections between CPUs and GPUs.

Some inference setups call for offloading important information like LoRA fine-tune weights and KV caches from previous inference calls to CPU memory, which is far larger than GPU memory. With Grace CPUs, this information can be retrieved much faster.

For the Rubin architecture, the Vera CPU (named for Vera Rubin) replaces the Grace CPU that was used on Hopper and Blackwell systems.

3.3 Instances

The atomic unit of GPU allocation on the cloud is an instance. An instance is a virtual machine that includes:

- **GPUs (device):** One or more GPUs for inference tasks.
- **CPUs (host):** General-purpose compute for any tasks that don't run on the GPU.
- **Memory (host memory):** General-purpose memory for CPU operations (traditional RAM).
- **Storage:** Disk memory for loading and storing large files.
- **Networking:** Physical network connections to the datacenter and eventually the public internet.
- **Interconnect:** Physical GPU-to-GPU and node-to-node connections for running on multiple GPUs at once.

Not all GPUs, and not all instances, are created equal. Depending on your cloud provider, instances vary in compute, memory, storage, and interconnect. While NVIDIA offers its own reference architectures, each cloud provider ultimately builds out systems according to their own preferences.

Even the GPU itself can differ from instance to instance. For example, NVIDIA A100 GPUs have two form factors: PCIe and SXM. But most inference on A100 runs on SXM GPUs as they have five percent higher memory bandwidth than the PCIe variant.

When provisioning instances, it's essential to understand exactly what you're getting. Any component of the instance, not just the GPU, could present a bottleneck or failure point during inference.

3.3.1 Multi-GPU Instances

Often, a model is too big to run on a single GPU, or inference engineers want to use multiple GPUs together to improve performance. It's common to need two, four, eight, or even more GPUs to run large models like DeepSeek or demanding modalities like video generation.

The standard unit for GPUs is a node, which contains eight individual GPUs. For example, a B200 node contains eight B200 GPUs.

These GPUs are connected together via two systems:

- **NVLink:** A one-to-one communication layer between GPUs, up to 1800 GB/s on Blackwell and 900 GB/s on Hopper.
- **NVSwitch:** An all-to-all communication layer on top of NVLink for coordination among all GPUs in a node.

These high-bandwidth interconnects make it possible to spread inference on a single model across multiple GPUs, up to a full 8-GPU node.

But sometimes one node isn't enough. Running extremely demanding inference workloads on more than eight GPUs requires a high-bandwidth interconnect between nodes.

The standard in node-to-node interconnect for NVIDIA GPUs is InfiniBand. InfiniBand competes with networking technologies like Ethernet, and in 2019 NVIDIA bought Mellanox, which manufactures InfiniBand.

InfiniBand is much slower than NVLink, with specs up to 400 Gb/s per Network Interface Controller (NIC). But it's the fastest node-to-node interconnect on the market – Ethernet maxes out at 100 Gb/s per NIC.

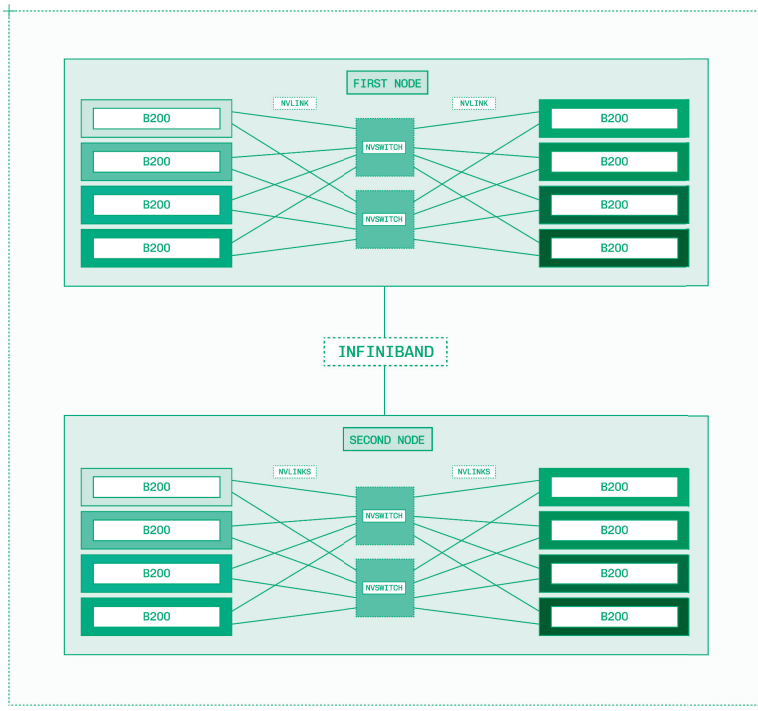


Figure 3.4: NVLink, NVSwitch, and Infiniband work together to enable GPU-to-GPU communication.

Not every cloud provider uses InfiniBand. Some offer their own interconnects, while others have InfiniBand on some GPUs but not all. When provisioning GPUs, double-check what interconnect is provided and what bandwidth that interconnect delivers.

In addition to InfiniBand, NVIDIA offers a high-end solution for NVLink connections among more than eight GPUs. Their NVL72 GB200 system combines 72 Blackwell GPUs and 36 Grace CPUs on a full-rack system. These systems provide massive throughput for serving the world's largest models with intense traffic.

The NVIDIA Vera Rubin NVL 144 CPX is the next generation of this massive system, with updated Vera CPUs and Rubin GPUs alongside the new Rubin CPX.

When working with multi-GPU and multi-node systems, keep relative bandwidths in mind. When an interconnect like NVLink is an order of magnitude faster than InfiniBand, it can handle far more data before becoming a bottleneck. Parallelism and disaggregation techniques discussed in sections 5.4 and 5.5 navigate this topology to deliver performant inference across multiple GPUs.

3.3.2 Multi-Instance GPUs

Sometimes, inference engineers run into the opposite problem: the GPU is too big for the model.

Using newer high-performance architectures like Hopper and Blackwell requires GPUs like the H100 or B200, with relatively large compute and memory allocations. For models with a couple of billion parameters or fewer, it's hard to utilize these GPUs effectively. Even with large batch sizes, valuable GPU resources are wasted.

Rather than running small models on older, lower-performance GPUs, there's a way to run these lightweight workloads on fractions of newer, high-performance GPUs.

Multi-instance GPU (MIG) is a hardware-level capability in larger GPUs including the A100, H100, H200, and B200. These GPUs can be split into as many as seven pieces. These fractional GPUs also receive a slice of CPU, RAM, storage, and other resources required to form an instance.

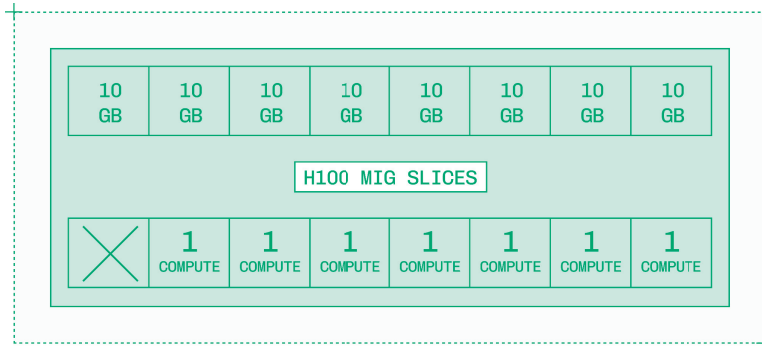


Figure 3.5: An H100 has eight memory slices and seven compute slices for assembling multi-instance GPU instances.

For example, an H100 MIG with three slices has about 3/7 of the available compute and can access as much as half of the total VRAM, or 40 GB. It also includes about half of the CPU cores, CPU memory, storage, and network bandwidth allocated to the underlying H100 to form an instance.

Software engineering generally works in multiples of two, so seeing seven compute slices may appear strange. Compute slices are made up of Streaming Multiprocessors in a GPU. GPUs generally do not have a clean multiple-of-two SM count, for example, an SXM H100 GPU has 132 SMs. Accordingly, seven evenly-sized compute slices are created, and the leftover SMs are left idle.

For small models like Orpheus TTS, a 3B parameter model, using two MIG instances is sometimes a more efficient use of resources than allocating the full GPU to a single instance.

3.4 Other Datacenter Accelerator Options

NVIDIA's leadership in the AI hardware market has made it the world's most valuable company. But it is far from the only company to make hardware that can run AI inference.

From fellow industry giants like Amazon and Google to a massive crop of startups, competitors are pouring billions of dollars into developing and manufacturing alternatives to NVIDIA GPUs.

While this book focuses on optimizing inference on NVIDIA GPUs, here's a short list of the other notable hardware options for running model inference.

Company	Stage	Flagship Product
AMD	Public	**MI350 GPU:** A datacenter GPU with competitive specs on AMD's own software stack.
AWS	Public	**Inferentia** and **Trainium:** A pair of chips purpose-built for inference and training, respectively, on AWS.
Cerebras	Startup	**WSE-3:** A wafer-scale chip with extremely high memory bandwidth to remove decode bottlenecks.

Company	Stage	Flagship Product
Etched	Startup	**Sohu:** An Application-Specific Integrated Circuit (ASIC) for the transformer architecture.
Furiosa	Startup	**RNGD:** A power-efficient accelerator designed for tensor contraction operations.
Google	Public	**TPU:** A Tensor Processing Unit is an AI-specific ASIC built for inference and training.
Groq	Startup	**LPU:** A composable language processing unit relies on SRAM for high memory bandwidth
Qualcomm	Public	**Cloud AI 100 Ultra:** A full-sized GPU composed of multiple power-efficient mobile GPUs.
Sambanova	Startup	**RDU:** A Reconfigurable Dataflow Unit with large memory allocation for trillion-parameter models.

GPUs are fairly general-purpose accelerators. Every hardware company competing to win inference workloads from NVIDIA is betting on a specific edge where their product can win:

- **Memory bandwidth:** Startups like Cerebras and Groq achieve high token per second scores for LLMs by accelerating decode on ultra-high-bandwidth memory.
- **Power efficiency:** Companies like Furiosa and Qualcomm design chips for lower power consumption, which leads to cheaper operating costs.
- **Platform integration:** Enterprises like Amazon and Google build deep integrations with their cloud service platforms and proprietary closed models.

While each of these accelerator options have their winning use cases, they share common challenges:

- **Software:** Without CUDA, hardware providers have to rebuild the entire inference stack for their accelerators.
- **Manufacturing:** Companies need to assemble the most complex object humankind has ever created.
- **Distribution:** After manufacturing a chip, providers need it installed and brought online to put capacity on the market.

Competition accelerates innovation. A market with more hardware options in the datacenter is good for every inference engineer. Competition in this space will only grow more robust as inference workloads become more valuable, as will exploration of options outside the datacenter such as local inference.

3.5 Local Inference

Local inference, also called edge inference, client-side inference, or on-device inference, means running AI model inference directly on the end user's device rather than on a centralized server.

Client-side inference has four massive advantages over server-side inference:

- **Zero network latency:** There's no communication overhead, saving tens or even hundreds of milliseconds.
- **Independence:** There's no dependency on internet connection or impact from high server traffic or downtime.
- **Improved privacy:** The end user's data never leaves their device.
- **Cost:** Datacenter GPUs are expensive, while edge inference is free for the developer, unlocking new business models.

Local inference sounds perfect in theory. In practice, there's a reason most inference happens in the cloud. There are four weaknesses to local inference that limit its applications:

- **Hardware capabilities:** Even high-end prosumer desktops offer a fraction of the speed and power of datacenter GPUs.
- **Thermal constraints:** Local devices have worse cooling than datacenters, further limiting their speed and power.
- **Fragmented support matrix:** Endless combinations of hardware and software make standardization challenging.
- **Battery life:** Inference is a demanding workload that quickly drains the batteries of laptops and smartphones.

When building with local inference, it's important to keep your audience in mind. An AI enthusiast may have the latest and greatest phone and a

powerful computer, but a median user is more likely to have older devices with less powerful components.

Local inference is turning the corner from experimentation to production, with a strong ecosystem across hardware and software and a vibrant community closely affiliated with the world of open models.

3.5.1 Desktop Inference

The classic local device is a workstation or gaming PC equipped with one or two high-end consumer GPUs from NVIDIA or AMD. While researchers and enthusiasts do use these setups, they're a small portion of the desktop inference market.

Increasingly, Apple is the leader in the desktop inference market. Apple's custom M-series CPUs and GPUs draw from a single unified memory, giving inference on GPUs access to far more memory, albeit at slower speeds.

The highest-end options from Apple and NVIDIA currently available on the market illustrate the tradeoff between memory capacity and speed.

Hardware	**NVIDIA RTX 5090**	**Apple M3 Ultra**
Memory	32 GB	512 GB
Bandwidth	1,792 GB/s	819 GB/s
Cost (full computer)	$5,000	$10,000

This trend continues through midrange hardware at more reasonable price points. Low-end computers like Chromebooks are not equipped to run any meaningful local inference.

The enthusiast-led open-source ecosystem focuses on running frontier open models on desktops and laptops. Today, running aggressively quantized 100B+ parameter models on high-end personal hardware is possible using tools like Ollama and llama.cpp.

The increased popularity of Mixture of Experts is also a tailwind for desktop inference. These models have fewer active parameters, meaning that

an individual user's single request only touches a fraction of the model's total weights.

Image generation is also quite popular on personal computers, especially via ComfyUI, a tool for assembling multiple image model components together into a single pipeline.

Smaller language models and other modalities like speech are also feasible to run on midrange computers, and the industry is rapidly developing browser inference libraries like WebLLM and other cross-platform standards to bring these capabilities from early adopters to the mainstream.

3.5.2 Mobile Inference

Local inference on mobile devices represents the majority of on-device workloads today. Both major operating systems offer tooling for developers to add edge inference to applications:

- **Android:** Google's AI Edge SDK and ML Kit GenAI APIs interface with Gemini Nano and OSS Gemma models.
- **iOS:** Apple's Foundation Models and Core ML frameworks provide APIs for models across modalities.

Mobile devices have extremely limited hardware capabilities and battery capacities, making inference even more challenging. Even high-end phones struggle to run models with more than one or two billion parameters.

Still, some modalities are well-suited for edge inference on phones. For example, transcription and speech synthesis models are latency-sensitive, and some models are small enough to run in real time on modern phones. Other discrete tasks, like translation, can be handled on edge devices by small fine-tuned models.

Like any other software, the future of inference isn't local or cloud, it's both working together to power fast and seamless user experiences. Small models and quick queries will run on end-user devices, while more demanding workloads will remain on datacenter GPUs in the cloud.

CHAPTER 4

Software

Software

NVIDIA's market dominance in the inference space is in no small part due to the robust and mature software ecosystem around its hardware.

Hardware iteration cycles are slow. Best-in-class hardware companies like Apple and NVIDIA release new architectures and generations at most yearly, with two-year release cycles being more common.

But software iteration is fast. Often, to run a newly released open model on day zero, you need to install a nightly build or other prerelease version of each of your software dependencies just to get support for the new model.

Software's fast iteration cycle and lower barrier to entry dramatically expands the landscape of inference engineering. While hardware centers on NVIDIA and a few competitors, there are countless companies building software at various levels of the inference stack.

For inference engineers, these are some key players:

- **NVIDIA:** Invests heavily in its own sometimes-proprietary software ecosystem, from CUDA up to Dynamo.
- **Hugging Face:** Maintains a model registry for all open models plus `transformers` and `diffusers`.
- **The Linux Foundation:** Maintains hardware-agnostic projects like PyTorch and vLLM.
- **LMSYS Org:** Develops essential tools for inference and evaluation, most notably SGLang.

There are thousands more companies, universities, and research institutions making essential open-source contributions to inference.

The software space is too big and too fast to exhaustively document in any book, much less in a single chapter. Instead, this chapter presents foundational technologies with long-term relevance.

Throughout the chapter, technologies are presented with increasing levels of abstraction:

- **CUDA:** Direct communication to the GPU for explicit control over computations and memory (section 4.1).
- **Deep learning frameworks:** Abstractions over CUDA for training, exporting, and running neural networks in Python (section 4.2).
- **Inference engines:** Highly configurable PyTorch-backed inference for common architectures (section 4.3).
- **NVIDIA Dynamo:** Sits on top of inference engines to power large-scale deployments (section 4.4).

Most inference engineering today happens at the higher levels of abstraction, configuring and deploying inference engines and orchestrating inference across multiple GPUs. No matter what level of the stack you work at, it's essential to have a strong mental model for the adjacent levels of abstraction to guide your work.

4.1 CUDA

CUDA is how you write code to run on NVIDIA GPUs.

More formally, CUDA is NVIDIA's proprietary computing platform and programming model for executing parallel tasks on GPUs. Both "platform" and "programming model" are broad definitions; it's easier to look at CUDA in its component parts:

- **CUDA kernel:** A user-defined function that executes parallelized code on the GPU.
- **CUDA graph:** A directed acyclic graph (DAG) of kernels and other GPU operations for optimizing repeated workflows.
- **CUDA driver:** A low-level interface between the application and the GPU hardware to manage memory and execution.
- **CUDA runtime:** A developer-facing API for launching kernels and managing memory.

CUDA – which stands for Compute Unified Device Architecture, though the acronym is rarely expanded today – is the foundation for the entire generative AI ecosystem on NVIDIA GPUs.

CUDA is not a programming language. Instead, CUDA programs are specified in a programming language, most often C++, then compiled into separate CPU and GPU code by a compiler like `nvcc`.

A kernel is just a function that does some parallel computation. Whenever you see the phrase "CUDA kernel" you can replace it with "a piece of code written for NVIDIA GPUs."

For a "Hello, World!" kernel, consider these six lines of C++ code which take in an array of length `n` and double each element in the array.

```
__global__ void doubleArray(float *arr, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        arr[i] *= 2.0f;
    }
}
```

Figure 4.1: Example CUDA kernel doubles each value in an array.

Ordinarily, on a CPU, this function would run in linear time, with each element in the array being doubled sequentially. But on a GPU, thousands of elements can be processed simultaneously, making this function much more efficient.

Writing CUDA kernels shifts inference engineering from thinking about algorithms to thinking about implementations. For example, the traditional attention algorithm that is central to generative AI can be expressed in a few dozen lines of code. However, FlashAttention, which is the same mathematical operation, takes tens of thousands of lines of code to implement the algorithm in a more memory-efficient manner for a specific GPU.

4.1.1 CUDA Kernels for Inference

Writing CUDA kernels does not mean building from scratch.

The prior art to build upon predates CUDA by decades. BLAS (Basic Linear Algebra Subprograms), first implemented for Fortran in the 1970s, is a specification for common linear algebra operations from dot products to matrix multiplication.

cuBLAS, the CUDA implementation for BLAS, brings this specification to CUDA in the form of pre-built kernels for essential linear algebra operations. Similarly, cuDNN (CUDA Deep Neural Network) provides primitives for neural networks.

Within BLAS, the most frequently used operation in inference is GEMM (General Matrix-Matrix Multiplication). Every linear layer in a model uses matrix multiplication, and cuBLAS provides a strong starting point.

But you aren't limited to the cuBLAS implementation. As operations like GEMM are so essential for inference, you may need more fine-grained control. You might write different GEMM kernels for matrices of different shapes or to better run on specific GPU architectures.

CUTLASS is a template library that provides building blocks for writing high-performance kernels. For example, FlashAttention 3 uses CUTLASS. CuTe, another template library, introduces abstractions for tiled tensor operations for recent architectures. With tools like CUTLASS and CuTe, kernels can be written at a higher level of abstraction while retaining strong performance.

Another source for kernels is FlashInfer, a library with high-performance implementations of kernels for LLM inference, including many optimized attention kernels and fused sampling functions.

4.1.2 CUDA Kernel Selection

Most inference engineers will never need to write their own kernels. However, kernel selection – choosing the best kernel from a range of options – is an important part of inference optimization.

Kernel implementations are highly specialized. CUDA exposes low-level APIs for memory management and parallel computing, and kernel engineers make implementation decisions based on the exact specifications of the hardware they're writing for.

It's important to understand how closely tied kernel implementations are to specific hardware details. Kernels often have hard-coded values based on the memory bandwidth or the number or layout of Tensor Cores on a given GPU.

A kernel written for an H100 will likely fail to take advantage of the architecture and extra memory of a B200, while a kernel written for that B200 could be backwards incompatible with the previous-generation Hopper architecture. With each generation of GPUs, porting handwritten kernels to run optimally on the new architecture takes substantial engineering work.

Most kernel selection is automatic. Deep learning frameworks and inference engines have pre-configured kernels for various architectures, while PyTorch and TensorRT-LLM include automatic kernel selection in their compilation steps.

However, you might manually choose a few kernels for essential algorithms to speed up inference.

For example, most production-ready GEMM kernels come from cuBLAS. But when the DeepSeek lab released their updated version of DeepSeek-V3, they also released DeepGEMM, which provides more efficient GEMM kernels for running matrix multiplication in FP8 on the Hopper GPU architecture.

Manual kernel selection lets you insert a DeepGEMM kernel as a plugin to speed up a specific step in inference, like multiplying two matrices of precise dimensions. Just be careful to ensure compatibility. For example, if you were to upgrade to a B200 GPU, you'd have to either swap the kernel back, wait for Blackwell support in DeepGEMM (now supported as of the time of publication), or port the kernel yourself.

4.1.3 Reducing Memory Accesses with Kernel Fusion

Running two different kernels back-to-back on the same data results in wasted reads from and writes to memory.

As a simple example, imagine two kernels: `multiply_by_2` and `multiply_by_3`. If these kernels were run back-to-back, the sequence would be:

1. Read the input vector `[1, 2, 3]` from memory.
2. Run `multiply_by_2` on the vector.
3. Save the output vector `[2, 4, 6]` to memory.
4. Read the new input vector `[2, 4, 6]` from memory.
5. Run `multiply_by_3` on the vector.
6. Save the final output vector `[6, 12, 18]` to memory.

The inefficiency is clear: steps three and four comprise an unnecessary round trip to memory. During decode, the bandwidth-bound phase of LLM inference, an inference engine can't afford unnecessary reads from or writes to memory.

Kernel fusion is the process of taking two or more kernels and re-implementing them into a single kernel that handles both operations. In this example, the fused kernel would be `multiply_by_6` and the new order of operations would be:

1. Read the input vector `[1, 2, 3]` from memory.
2. Run `multiply_by_6` on the vector.
3. Save the final output vector `[6, 12, 18]` to memory.

In practice, kernel fusion is far more complicated – functions are more complex, and data overlap isn't as clean. But there are common patterns in kernel fusion for inference, like combining matrix multiplication, bias adding, and activation.

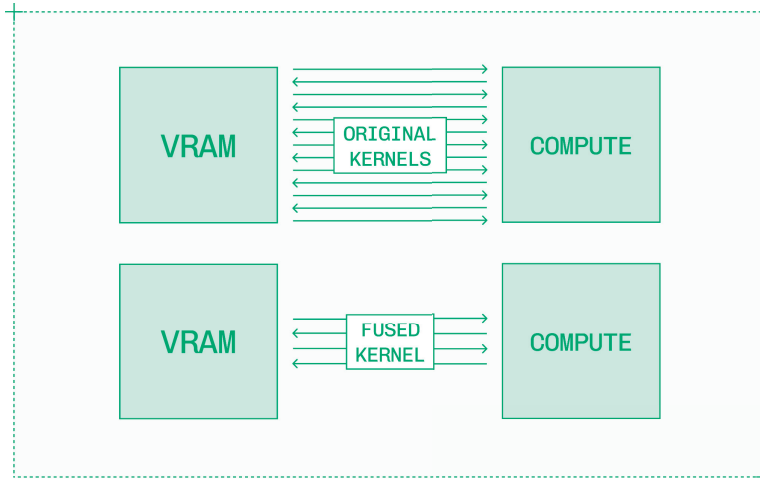


Figure 4.2: Kernel fusion reduces reads and writes between memory and compute within a GPU.

Kernel fusion can be an automatic or a manual process. Compilers can identify straightforward fusion opportunities and automatically create fused kernels. But more sophisticated algorithms, like FlashAttention, require handwritten fused kernels, which are used via plugins during inference.

4.2 Deep Learning Frameworks and Libraries

Deep learning frameworks and libraries are the bridge between working directly in CUDA and using off-the-shelf inference engines like vLLM. These libraries are used in both training and inference.

Over the past few years, PyTorch has emerged as the clear leader at this level of the stack. There are two other frameworks that, for concision, I will only mention briefly:

- **TensorFlow:** An end-to-end machine learning platform officially supported by Google, TensorFlow was prominent in the 2010s ML era but has fallen out of favor today.
- **JAX:** A research project unofficially associated with Google, JAX presents a simpler interface without as many legacy features and operations. However, as the documentation warns, expect sharp edges.

The remainder of this section focuses on PyTorch and the technologies around and above it in the stack.

4.2.1 PyTorch

PyTorch is a Python package for describing tensor operations. Originally created at Meta and now a part of the Linux Foundation, PyTorch is the industry standard technology underlying both training and inference for generative AI models.

Personally, I've been a Python programmer for my entire career, and I find writing low-level C++ difficult. With PyTorch, I can write highly performant inference code in Python for both CPUs and GPUs, but I always have the option to dip down into CUDA when necessary by plugging in specific kernels.

PyTorch can train any kind of neural network. The PyTorch documentation shows a basic neural network example:

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Figure 4.3: A basic neural network in PyTorch, adapted from the PyTorch documentation.

While this is a very simple example, you may recognize the linear layers and ReLU activation functions discussed in chapter 2 as key pieces of neural networks.

PyTorch automatically computes gradients for any differentiable function via its `autograd` module. This is what makes PyTorch so powerful for training – you define a computation graph, and you get a gradient to train against.

But PyTorch is special because it isn't just great for training, it's also powerful for inference. PyTorch balances built-in functions and automatic performance optimizations with manual control where you need it.

The step that transforms a model from training to inference is compilation. PyTorch compilation (`torch.compile`) targets a specific GPU and performs automatic kernel selection and kernel fusion to ensure optimal performance.

`torch.compile` can't fuse plugin kernels like DeepGEMM, FlashAttention, or custom kernels. This limits its utility for LLM inference, where most kernels are custom. However, PyTorch compilation is useful for optimizing less common model architectures and compiling long sequences of lightweight kernels. When you are optimizing a model that has a custom or rare architecture, you may have to rewrite functions to be more abstract – especially with respect to Python-specific language features – for compilation to succeed.

PyTorch alone is a powerful and flexible tool for building high-performance inference services. But there is a rich ecosystem on top of PyTorch that makes it faster to implement, compile, and execute optimized code for common model architectures.

4.2.2 Model File Formats

The dominant file format for serializing model weights is safetensors, created by Hugging Face.

Safetensors is a replacement for generic formats like bin designed specifically for holding model weights. The safety in safetensors comes from the fact that unlike a general format that can execute arbitrary Python code during deserialization, safetensors only hold tensor data, not executable code.

Generative AI models have hundreds of gigabytes of weights. These weights are split across dozens of safetensors files. The safetensors format uses memory mapping to ensure that the files are loadable without allocating the full memory, which makes loading model weights faster and safer.

Another leading format, ONNX (Open Neural Network Exchange), stores weights along with an execution graph for the model. Where the safetensors format separates the weights from the architecture, ONNX bundles them together.

ONNX files are highly portable. With deep integration into PyTorch and support for multiple hardware options, ONNX is a great alternative to safetensors for when you want to store model graphs, not just weights.

4.2.3 ONNX Runtime and TensorRT

ONNX Runtime and TensorRT are high-performance inference runtimes. PyTorch models can be exported to the ONNX format, which ONNX Runtime can execute directly or TensorRT can compile into a highly optimized engine.

ONNX Runtime	**TensorRT**
Open source, associated with the Linux Foundation	Mix of proprietary and open components, built by NVIDIA
First-class exporter in the PyTorch ecosystem	Integrated with PyTorch via Torch-TensorRT
Supports many types of GPUs	NVIDIA GPUs only

ONNX Runtime is an open community standard, while TensorRT is specific to NVIDIA GPUs.

The export process looks somewhat like Torch compilation. However, these standards do not support every data structure, type, and operation within PyTorch. The export process can identify these issues in PyTorch code, but this gets tricky with more complex models.

An example is DeepSeek V3, which introduces Multi-Latent Attention (MLA). MLA, as implemented in PyTorch, is difficult to export, but the

transformers architecture overall is simple enough that hand-fusing kernels is feasible.

Today, it's increasingly popular to skip directly from PyTorch to an inference engine like vLLM or TensorRT-LLM for models that these engines support, bypassing the intermediate representation step and exporting weights only as safetensors. ONNX Runtime and TensorRT are still widely used – TensorRT especially for its strong out-of-the-box runtime for image and video models – but the industry is bifurcating between the control of handwritten PyTorch code or the convenience of prebuilt inference engines.

4.2.4 Transformers and Diffusers

The `transformers` and `diffusers` libraries by Hugging Face are built on PyTorch but are not designed to run large-scale production inference. Instead, these libraries offer reference implementations of models for inference engineers to learn from and adapt.

While these libraries are toolboxes for tinkering, they do include essential information about models and useful utilities for building model inference servers. The `config.json` file that ships with models implemented for `transformers` and `diffusers` contains essential information, and the libraries' utilities for Hugging Face operations like downloading model weights are widely used.

You'll find `transformers` or `diffusers` sample code in the model card for most popular open models on Hugging Face. This sample code is great for understanding the exact input and output spec of a model or for running local inference and notebooks. But for production, you'll want to either write and compile PyTorch code directly or use a production-ready inference engine.

4.3 Inference Engines

There are three competitive inference engines on the market: vLLM, SGLang, and TensorRT-LLM.

These frameworks offer good out-of-the-box performance for LLMs and other modalities with similar architectures (chapter 6).

In late 2025, vLLM and SGLang also began supporting some image and video generation models via vLLM Omni and SGLang Diffusion, respectively. TensorRT-LLM does not support image or video generation models. Inference engineers can also use TensorRT or PyTorch directly to run these models (section 6.5).

Inference engines are powerful because they are configurable. Working with pre-optimized components at a higher level of abstraction, inference engineers can spend their time testing combinations of techniques rather than repeating routine implementations.

At a very high level, vLLM and SGLang are more general tools that are easier to adopt and have day zero support for more models, while TensorRT-LLM has a steeper learning curve but usually achieves the best performance.

Engine	**vLLM**	**SGLang**	**TensorRT-LLM**
Performance	Good	Good	Best
Ease of use	Easy	Easy	Hard
Model support	Most	Most	Some
Hardware	GPU, TPU	NVIDIA, AMD	NVIDIA only
License	Apache 2.0	Apache 2.0	Apache 2.0

Each framework runs out of the box with core features like continuous batching and supports the main performance optimization techniques – post-training quantization, speculative decoding, prefix caching, parallelism, disaggregation.

At Baseten, we use all three frameworks, though we use TensorRT-LLM the most. Inference engineers should be familiar with all three and select on a deployment-by-deployment basis.

4.3.1 vLLM

vLLM has the largest market share among inference engines. GitHub stars are a rough measure for popularity, but at the time of publication vLLM has twice as many stars as SGLang and TensorRT-LLM combined.

First released in the summer of 2023, vLLM is the oldest of these inference engines by a few months. Originally created at UC Berkeley, vLLM is now hosted by The PyTorch Project within The Linux Foundation.

vLLM's best selling point is its broad support. It supports the most hardware options – NVIDIA, AMD, and Intel GPUs along with Google TPUs – as well as the most models and architectures. Just about every open LLM out there integrates with vLLM from day zero. vLLM also supports multimodal inference via vLLM Omni, which extends the engine to support image, audio, and video inputs and outputs.

One of the core principles of inference engineering is that the more constraints you can introduce, the better performance you can achieve. vLLM's broad platform can achieve impressive performance results when properly configured, but in my experience it falls short of the highest-end performance possible with narrow frameworks like TensorRT-LLM.

vLLM's developer experience is built around the `vllm serve` command, with server configuration passed in as flags.

```
vllm serve deepseek-ai-DeepSeek-V3.1
  --enable-expert-parallel
  --tensor-parallel-size 8
  --served-model-name ds31
```

Figure 4.4: vLLM inference example on eight GPUs.

vLLM is pip-installable and provides official Docker images with pre-bundled dependencies and support for various hardware architectures.

You should use vLLM when:

- You want to quickly stand up a model server that will offer solid performance out of the box for almost any open model.
- You want to run an "Omni" model with multiple input and output modalities.

- You are using a smaller GPU or older architecture where TensorRT-LLM offers few performance benefits.

4.3.2 SGLang

SGLang is the other major community-driven fast inference framework. First released in December 2023, SGLang has risen to prominence alongside Chinese open models like DeepSeek and Qwen and is the engine of choice for inference at xAI.

SGLang's unique angle on the problem of model serving is expressed in its developer experience, which pairs a fast backend runtime with a flexible frontend language. In practice, that means you can choose individual components of your engine for deep customization without needing to rewrite everything else from scratch.

SGLang supports both NVIDIA and AMD GPUs, and has strong day-zero support for a wide range of models. SGLang works closely with labs like DeepSeek, Qwen, Kimi, and Z AI to release optimized implementations of new architectural features like DeepSeek's Multi-Latent Attention.

SGLang has invested heavily in supporting large-scale deployments of MoE LLMs, specifically multi-node deployments on systems like GB200 NVL72 for high throughput. These systems offer extremely cost-efficient inference for large models with significant traffic.

SGLang's developer experience is built around the `sglang.launch_server` command, with server configuration passed in as flags.

```
python3 -m sglang.launch_server
  --model-path deepseek-ai/DeepSeek-V3-0324
  --speculative-algorithm EAGLE
  --speculative-num-steps 1
  --speculative-eagle-topk 1
  --speculative-num-draft-tokens 2
  --trust-remote-code
  --tp 8
```

Figure 4.5: SGLang inference example on eight GPUs.

SGLang also supports image and video generation model inference via SGLang Diffusion.

SGLang Diffusion introduces a pipeline abstraction which orchestrates a number of stages. This flexible approach maps closely to the architecture of image and video generation models. For performance, SGLang Diffusion adds support for various diffusion-specific parallelism methods and re-uses the scheduler and optimized kernels from the main SGLang package.

You should use SGLang when:

- You want excellent out-of-the-box throughput with decent latency on large MoE models like DeepSeek and Kimi.
- You want the inference engine experience for image and video generation models.
- You want control and customization and are excited to participate in the SGLang community.

4.3.3 TensorRT-LLM

TensorRT-LLM is NVIDIA's open-source inference engine. Of the three main options, TensorRT-LLM offers the highest performance and the most flexibility to expert users.

A note on naming: There are two major versions of TensorRT-LLM. Only the older version is actually related to TensorRT:

- **TensorRT-LLM V0 (0.X.Y):** Major versions starting with zero are a plugin for NVIDIA TensorRT.
- **TensorRT-LLM V1 (1.X.Y):** Major versions starting with one are a standalone package based on PyTorch with no dependency on TensorRT.

Originally, TensorRT-LLM built a TensorRT engine for serving language models. With the modern PyTorch-based version, TensorRT-LLM bypasses the intermediate representation of TensorRT and uses PyTorch directly.

TensorRT-LLM V1 was released in the summer of 2025. Deployments of the previous major version are still common – always be sure to check which version you are using.

TensorRT-LLM achieves the best performance in large part because it has access to kernels written by NVIDIA engineers, including some closed-source kernels. These handwritten and manually fused kernels offer excellent support for the latest hardware architectures like Hopper and Blackwell and for NVIDIA-specific number formats like NVFP4.

TensorRT-LLM offers a robust implementation of in-flight batching (token-level continuous batching), which helps with throughput. It also supports just about every model performance optimization setting you could ask for, including quantization, speculation algorithms, prefix caching, chunked prefill, flexible parallelism, and disaggregation.

With V1, TensorRT-LLM introduces a developer experience that looks a lot like vLLM and SGLang. However, in addition to flag arguments on the `trtllm-serve` command, it expects a `config.yaml` file for deeper customization.

```
trtllm-serve deepseek-ai/DeepSeek-R1-0528
  --host 0.0.0.0
  --port 8000
  --backend pytorch
  --max_batch_size 1024
  --max_num_tokens 3200
  --max_seq_len 2048
  --kv_cache_free_gpu_memory_fraction 0.8
  --tp_size 8
  --ep_size 8
  --trust_remote_code
  --extra_llm_api_options ${EXTRA_LLM_API_FILE}
```

Figure 4.6: TensorRT-LLM inference example on eight GPUs.

The best way to install TensorRT-LLM is by running it via one of NVIDIA's official Docker containers

Use TensorRT-LLM when:

- You are running a well-supported model architecture on a Hopper or later GPU.
- You are willing to do extra engineering work to get the best possible performance.
- Optionally, you are planning to use NVIDIA Dynamo for serving and want the most deeply integrated engine.

4.4 NVIDIA Dynamo

NVIDIA Dynamo is a distributed system for model serving first announced at NVIDIA GTC in March 2025.

Dynamo works with every inference engine – vLLM, SGLang, and TensorRT-LLM – as backends, with Dynamo itself providing an orchestration layer for large-scale deployments.

Dynamo provides support for essential model performance techniques:

- **KV cache re-use:** Retaining KV information between requests and routing requests based on prefix match.
- **Disaggregation:** Separating prefill and decode onto individually optimized engines with independent scaling.
- **Multi-node parallelism:** Optionally using two or more nodes of GPUs in a single replica for a model, usually with Expert Parallelism.

Each of these techniques will be detailed in chapter 5. As with the inference engines, there is a lot of work for inference engineers to do to configure Dynamo for their use case and achieve maximum performance.

Dynamo's thoughtful abstractions for distributed KV routing, disaggregation, and multi-node model parallelism provide high-performance aggregation of information during runtime, allowing for real-time adjustments to configuration as traffic fluctuates. For example, you can automatically scale up and down prefill and decode workers with an SLA-based planner operating on user-defined TTFT and TPS constraints.

As a general principle, the more scale you have, the more tools and techniques there are available to you for inference optimization.

Dynamo is built for scale: big models, big traffic. It excels at serving foundation models like the trillion-parameter Kimi family to a large number of concurrent users. For smaller models, Dynamo can still offer moderate performance improvements on large-scale deployments.

If you're building an inference API for a built-from scratch foundation model or serving an open model in a high-usage product, Dynamo is a great choice.

But many deployments don't need the additional complexity of Dynamo. Unless you're operating with enough volume for disaggregation and KV-aware routing to matter, Dynamo will be unnecessary work and excess overhead. In these cases, you can use inference engines directly.

Dynamo is the newest project covered in this chapter, and features are still being built out. Dynamo is open source under the Apache 2.0 license. The community around Dynamo is active, and the project welcomes contributions with a public CI and support from NVIDIA engineers.

4.5 Performance Benchmarking and Load Testing

Benchmarking is an essential part of model performance optimization. Without precise, accurate performance benchmarks, there's no way of knowing if your optimizations are actually working.

A high-quality benchmark simulates real life as closely as possible. The best benchmark is to shadow real-world production traffic onto the system you are testing. Shadowing is the process of copying incoming requests onto the test system so that you can benchmark its performance without affecting the original request.

If you can't shadow real usage, you'll need to simulate it. LLM performance is affected by a number of factors. When simulating traffic, you need to match your expected production workload on multiple dimensions:

- **Sequence lengths:** Time to first token and memory usage rely on the input sequence length (ISL) and output sequence length (OSL), meaning the number of tokens in the prompt and response.
- **Volume and pattern of traffic:** Batching and server load depend on the number of concurrent requests. Jitter traffic to mimic real usage.
- **Request contents:** The actual prompt within each request affects performance factors like cache hit rate and draft token acceptance.
- **Input parameters:** Settings like temperature and reasoning effort that affect inference should be set to their anticipated production values.

Remember, optimization is about tradeoffs and constraints. If you're maximizing benchmark performance against bad inputs, performance in production won't match expectations.

4.5.1 Performance Benchmarking Tooling

As a performance benchmark should closely reflect production traffic, everyone's benchmarking setup should look a bit different. But there are a few common tools:

- **SGLang Genai-bench:** A CLI and dashboard by the SGLang team for benchmarking models deployed with any inference framework.
- **NVIDIA GenAI-Perf:** A client-side tool by NVIDIA for measuring latency and throughput on varied traffic.
- **Locust:** An open-source load-testing tool, not specific to generative AI systems, that simulates as many as millions of simultaneous users.

Another great tool for benchmarking is open-source evals datasets – from general evals like MMLU and gsm8k to domain-specific evals like SWE-bench.

While the purpose of benchmarking work is to measure performance, not model output quality, these eval datasets serve two purposes: acting as a set of varied and realistic inputs, and spot checking that performance optimizations haven't impacted model output quality.

When possible, choose an eval dataset that matches the expected use of your production system, like HumanEval when reducing latency for a code completion system.

4.5.2 Performance Benchmarking Tips

Along with being realistic, great benchmarks are also consistent. Make sure your benchmarks send enough traffic to get a good read on performance without being swayed by outliers. When in doubt, run a benchmark multiple times and average the results.

Before you do any performance optimization work, start with a solid baseline benchmark. As you test optimizations, keep a consistent configuration in your benchmarking setup, and test each optimization individually as well as collectively to fully understand what is driving performance improvement. In some cases, optimizations can work against each other, like trying to run speculative decoding with large batch sizes.

The principle of changing one thing at a time applies to your benchmarking configuration as well. It's common to need to test various traffic patterns or sequence shapes, but as with any experiment only change one variable at a time to ensure that you are getting clear results.

4.5.3 Profiling Performance

Profiling is one click deeper than benchmarking. Where a benchmark gives a single figure (e.g., the P90 TTFT is 350 ms), a profiling tool shows where each of those milliseconds was spent in the inference process. Benchmarking tells you how your system is performing; profiling tells you why it's performing that way.

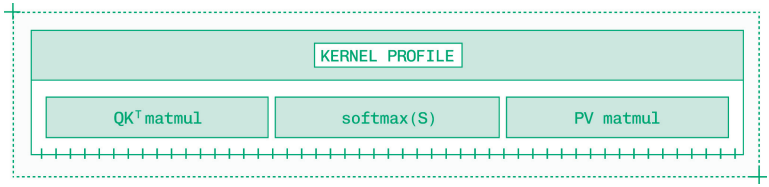


Figure 4.7: A kernel profiler shows you how long each operation within a kernel takes to execute, revealing bottlenecks.

Most inference engineers won't need to do profiling as part of their daily work. When using an already high-performance tool like the inference

engine TensorRT-LLM, your workflow is a cycle of configuration and benchmarking – profiling would be extraneous.

However, if you're contributing to an inference framework like vLLM or SGLang, writing your own inference service in PyTorch, or operating at the cutting edge of a new modality like video generation, performance profiling should be part of your toolkit.

The most popular profiling tools for inference are:

- **PyTorch Profiler**: An easy-to-use profiling library for capturing step-by-step performance metrics (CPU time, GPU time, memory usage) during inference.
- **NVIDIA Nsight Systems (NSys)**: A featureful but complex tool for GPU and CPU sampling and tracing that provides system-wide analysis across multiple GPUs and their interconnects.
- **NVIDIA Nsight Compute (NCU)**: A profiling utility and CLI for in-depth analysis of individual CUDA kernels on both compute and memory usage.

In addition, frameworks like TensorFlow and TensorRT ship with their own built-in profilers.

Profilers are valuable because they give you granular information about compute and memory usage, which guides your optimization work toward improving the most expensive steps in your inference pipeline.

For example, using PyTorch Profiler you might find that activation functions are taking an unusually long time due to excess memory reads, and figure out how to write a fused kernel that runs activations alongside attention to prevent the excess reads. Then, you would insert that new kernel into your PyTorch code and re-run system-level benchmarks to see if you've achieved your latency targets.

Together, profiling and benchmarking give you the information you need to improve system performance and, eventually, the confidence to deploy your optimizations in production.

CHAPTER 5

Techniques

Techniques

One of the coolest parts about working in inference engineering is that unlike many industries where new academic research takes years or decades to be adopted by industry, techniques from new papers are live in production within months or even weeks.

There is a gap to cross between research and production, and some of the most visible inference engineering work in the industry comes from bridging that gap.

A core principle of inference engineering is that the more constraints you can introduce in your inference system, the better performance you'll achieve. This principle continues to apply throughout this chapter, with techniques like disaggregation, which allows you to constrain individual engines to prefill and decode.

With these model performance techniques, there's a new principle to keep in mind: the more traffic you have, the more performance optimizations you can make (while keeping unit economics reasonable). Higher model parallelism across more GPUs, KV-aware routing, and dynamic disaggregation only make sense when you have a large number of GPUs, often multiple nodes, serving the same model with vertical scale and horizontal replication.

Real-world traffic defies constraints. But with volume, you can adapt your systems over time to match the changing nature of usage. Tuning the parameters of inference engines, speculation algorithms, and model servers isn't a one-time task. Instead, either through iterative deployments or dynamic runtime adjustments, you can continuously improve the performance of your inference system.

Finding the right combination of techniques and configurations takes patient experimentation. I remember an internal hackathon during which one of Baseten's inference engineers was working on an autocomplete model for code and ended up trying 77 different configurations via a hand-written script before finding a non-obvious solution that doubled TPS for a customer's model.

To make inference optimization even more complex, sometimes techniques are symbiotic and sometimes they are incompatible. For example, quantizing the KV cache alleviates a bottleneck in disaggregation, but increasing batch sizing reduces the compute available for speculation. An inference engineer's goal is always to create a balanced set of optimizations that delivers more than the sum of its parts.

This chapter introduces five key categories of applied research for inference acceleration: quantization, speculation, caching, parallelism, and disaggregation. In each section, pay special attention to the recommended circumstances for using each technique and the potential bottlenecks or tradeoffs each introduces.

5.1 Quantization

Quantization improves latency (both TTFT and TPS), increases system throughput, and opens up headroom for other optimizations like disaggregation, speculation, and prefix caching to be even more effective. But when it goes wrong, quantization can materially reduce the model's output quality.

Models are trained with weights, activations, and other components represented in a certain native number format. Usually, this is BF16 or FP16, though 8-bit and 4-bit native precisions are becoming more popular in training.

Post-training quantization works by changing those model weights and other values from their native number format to a lower-precision format. Cutting precision in half improves performance in both phases of inference:

- **Prefill:** Compute-bound prefill now runs on lower-precision Tensor Cores with twice the FLOPS.
- **Decode:** Memory-bound decode now loads half as much data per value, effectively doubling memory bandwidth.

Working with quantized data does introduce overhead, so it's not linearly twice as fast to go from 16 to 8 bits. In practice, quantization down a single level of precision generally offers 30 to 50 percent better performance for LLMs.

The catch with quantization is that it runs the risk of reducing the model's output quality. Quantization has the potential to introduce precision errors throughout the calculations that power inference.

Precision errors compound over time. Consider what happens when you square and cube different precisions of Pi:

Pi precision	Pi squared	Pi cubed
3.14159	9.869588	31.006198
3.14	9.8596	30.959144
3	9	27

Most of the work in quantization is around both preventing precision errors and minimizing their impact on the final model output.

5.1.1 Number Formats

Quantization introduces a new collection of essential terms and abbreviations. The most important ones to know are the common number formats:

Name	Abbr	First architecture
64-bit Floating Point	FP64	Fermi (2010)
32-bit Floating Point	FP32	Kepler (2012)
16-bit Floating Point	FP16	Pascal (2016)
Brain Floating Point 16	BF16	Ampere (2020)
8-bit Floating Point	FP8	Hopper (2022)
Mixed-Precision FP8	MXFP8	Blackwell (2024)
8-bit Integer	INT8	Pascal (2016)
6-bit Floating Point	FP6	Blackwell (2024, experimental)
4-bit Floating Point	FP4	Blackwell (2024)
Mixed-Precision FP4	MXFP4	Blackwell (2024)
NVIDIA FP4	NVFP4	Blackwell (2024, proprietary)
4-bit Integer	INT4	Turing (2018)

The largest number format, FP64 or "double precision," is only used for high-precision scientific computing, not AI training or inference. FP32 is sometimes used for training, but almost never for inference. FP6 is more experimental at the time of publication, though AMD GPUs are rapidly adopting the format.

That leaves 16, 8, and 4-bit precisions as the primary formats for inference. Number formats have a:

- **Precision:** The number of bits used to express a single value in the format. For example, FP16 uses 16 bits.
- **Type:** Whether these bits are interpreted to represent an integer (no decimal) or a floating-point number (a decimal).
- **Scale factor:** A multiplier used to map values from a low-precision format back to the higher-precision format.

Together, these attributes determine the two factors behind how well a number format represents values used in inference:

- **Dynamic range:** The difference between the lowest and highest value that can be represented in the format.
- **Granularity:** The number of parameters or other values that are quantized along a single scale factor.

Dynamic range is essential to low-precision inference without quality loss. 16 bits can represent 65,536 distinct values, while 8 bits can only represent 256 different values. The dynamic range is the distribution of these values – the difference between the smallest and largest available value.

Dynamic range explains why floating-point formats are better than integer formats for inference. Floating-point formats have three properties:

- **Sign:** A single bit that represents whether the number is positive or negative.
- **Exponent:** A set of bits that, taken together, represent an exponent factor.
- **Mantissa:** A set of bits that, taken together, represent the base value multiplied by two to the exponent.

An FP8 number in a E4M3 data format means it has a 4-bit exponent and a 3-bit mantissa, with the remaining bit for the sign. Integer formats only have sign and value bits.

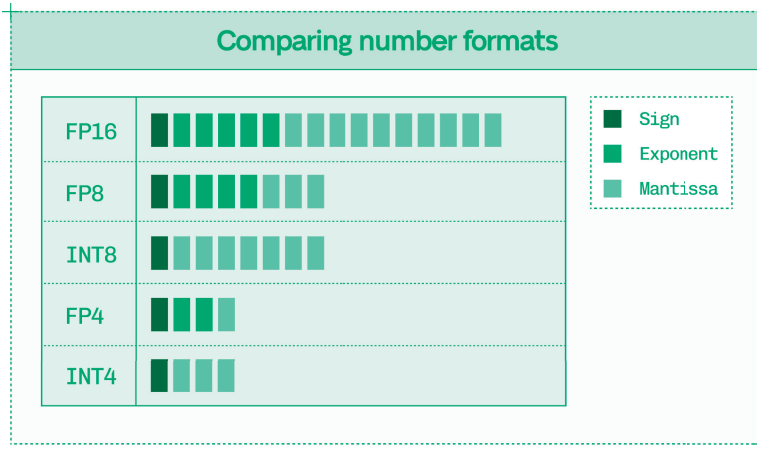


Figure 5.1: Floating-point number formats have both exponent and mantissa bits along with the sign bit.

The exponent in floating-point numbers gives it a higher dynamic range, meaning it can better express very large and very small numbers. This is important because outlier values are significant in inference, and floating-point number formats better represent outliers after quantization.

Within floating-point formats, there are multiple options at each precision, like FP4, MXFP4, and NVFP4. These formats differ in granularity, or the number of values that are quantized by a single scale factor.

Quantization can be applied at the:

- **Tensor level:** Calculate a single scale factor for the entire QKV tensor.
- **Channel level:** Calculate a different scale factor for each feature vector within the tensor.
- **Block level:** Within each feature vector, divide the vector into blocks of N values and calculate a scale factor for each block.

More granular quantization has a lower chance of smoothing over outliers, preserving quality. However, more granularity introduces more overhead for storing and applying scale factors.

MXFP8 and MXFP4, the new number formats supported by Blackwell, are "microscaling" formats that compute a blockwise scale factor on every 32 parameters, reducing the impact of these number formats' lower dynamic range.

NVFP4, a 4-bit format by NVIDIA, offers even higher granularity than the MX formats with a block size of 16 and a secondary 32-bit global scale factor to further combat the quality loss that 4-bit formats introduce.

The tradeoff to a microscaling format is that the small-block scale factor must also be stored in memory, slightly reducing the performance gains from quantization. Additionally, both the tensor and block scale factors need to be applied, introducing a bit of compute overhead. Blackwell GPUs offset this overhead via scale factor application in Tensor Cores.

While the focus of this book is on inference engineering in the datacenter, quantization is an essential topic for local and edge inference, especially for large models. GGUF, a binary format for storing models, is the most popular choice for distributing highly quantized models on Hugging Face, with individual researchers and companies squeezing huge models like DeepSeek onto consumer hardware like Apple computers.

These quantization strategies combat quality loss through dynamic quantization, where certain layers or other components of the model are left in their original precision, while others are quantized to integers with as little as one bit of precision. Dynamic formats represent their average precision, which is why you might see something like fine-tuning lab Unsloth's popular 1.58-bit quantization.

While these dynamic quantizations are impressive feats of engineering and are great for local inference, inference engineers working on production systems should stick with floating point number formats – integer formats are not suitable for quality-sensitive workloads due to their lack of dynamic range.

Instead, 8-bit floating-point formats (FP8, MXFP8) are generally the sweet spot for improving performance without sacrificing quality. FP4 is promising, especially the NVFP4 format which introduces a higher level of granularity for improved accuracy, but FP8 and MXFP8 provide the most flexibility, especially when quantizing the KV cache.

5.1.2 Quantization Approaches

The more parameters a model has, the less sensitive it is to quantization as each individual parameter is less important. However, even for very large models, it is essential to quantize carefully.

Quantization can happen during or after training:

- **Quantization-aware training:** Training weights and computing scale factors together to ensure that the final converged weights are accurate at a given precision.
- **Post-training quantization:** Converting finished model weights to a new precision by computing scale factors and preserving accuracy via calibration.

While some labs release models created with quantization-aware training, like GPT-OSS in MXFP4 and Kimi K2 Thinking in INT4, inference engineers working with open models only have the ability to perform post-training quantization as they are working with finished weights.

A leading tool for post-training quantization is NVIDIA TensorRT Model Optimizer (ModelOpt), an open-source library that also supports pruning, distillation, and sparsity. ModelOpt outputs are compatible with all inference engines (vLLM, SGLang, TensorRT-LLM).

After picking a precision, there are two decisions to make before doing post-training quantization:

1. What parts of the model (weights, activations, KV cache, attention) should be quantized?
2. What number format offers the appropriate dynamic range and granularity?

These decisions turn quantization from a binary choice into a spectrum of tradeoffs around performance and quality.

Components of a model have varying sensitivity to quantization. Reducing the precision of more sensitive components runs a higher risk of quality degradation. From least to most sensitive:

1. **Weights:** Specifically the linear layers are least sensitive to quantization.
2. **Activations:** The intermediate output of activation functions are only somewhat sensitive to quantization. Note that the activation functions themselves are rarely quantized as they are such a tiny fraction of the model's weights.
3. **KV cache:** The cached values from the attention calculation are moderately sensitive to quantization.
4. **Attention:** The attention layers of a model are highly sensitive to quantization, especially equations like softmax.

Within each of these components, you can get more selective about quantization.

Even in linear layers and activations, which are generally the least sensitive to quantization thanks to their size, early and late layers like the input and output layer of the neural network may be left in their original precision as these layers are more sensitive.

While quantizing weights and activations helps performance directly, KV cache quantization gives an additional boost to techniques like prefix caching and disaggregation. The KV cache is a valuable resource. Quantizing it allows inference engines to store more of it in memory and read it more quickly.

However, the KV cache for each token is used by each subsequent token. This means precision errors introduced by quantization can compound from token to token.

Compounding errors is exactly the reason why attention layers are the riskiest to quantize. Not only is attention very sensitive to dynamic range, but each attention calculation relies on the results of each previous atten-

tion calculation. Over a sequence of thousands of tokens, these errors accumulate quickly.

All but the most aggressive quantization schemes run functions like softmax in their original precision.

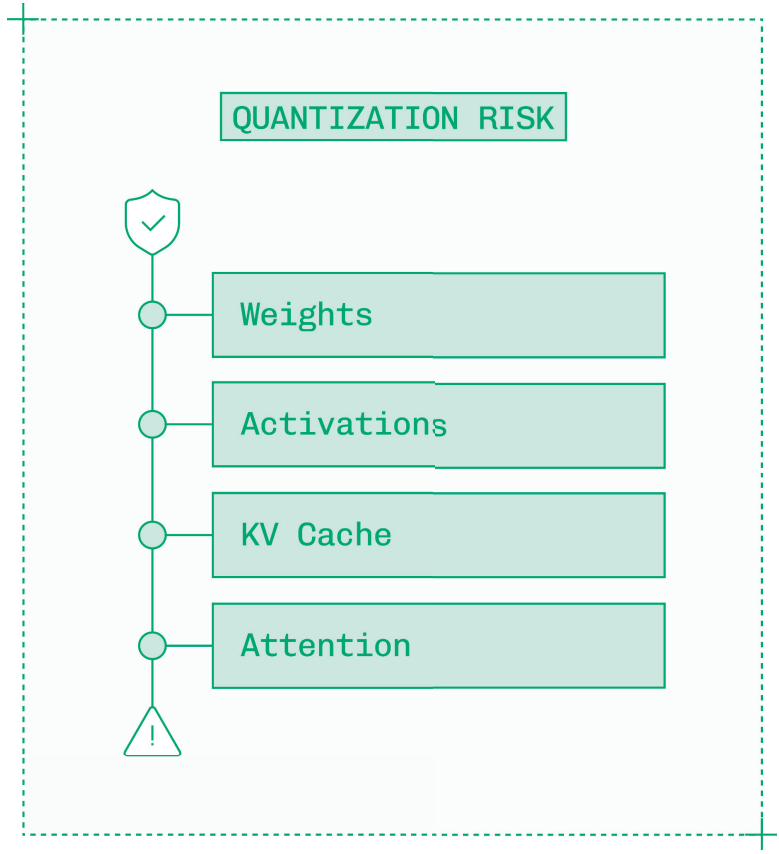


Figure 5.2: Quantization risk is low for weights and activations, moderate for KV cache, and high for attention.

A moderate approach to low-precision inference uses a format like FP8 with high dynamic range – if possible, a microscaling format like MXFP8 – to carefully quantize select linear layers, activations, and often KV cache values. Even with these high dynamic range formats, components of the attention layer are rarely quantized.

5.1.3 Measuring Quality Impact

The standard for production-ready quantization is zero perceptible quality loss. After quantizing a model, it's essential to thoroughly test its output quality versus the original precision.

There are three methods for checking model quality after quantization:

1. **Perplexity:** Calculate the perplexity score for the quantized model and compare with the original.
2. **Intelligence benchmarks:** Run a standard intelligence benchmark like MMLU or SWE-bench and compare to original scores.
3. **Custom evals:** Run a product-specific evaluation suite on the quantized model and compare to the original weights.

In every case, you're looking for a difference in scores that's indistinguishable from noise. LLMs are non-deterministic, so scores vary slightly from run to run.

The simplest check on quality is perplexity. Rather than asking the model to generate output, perplexity gives the model expected output sequences and calculates the likelihood of the model predicting those tokens.

A higher perplexity means a model is more "surprised" by the sequences – not what you want from a model that's supposed to predict tokens. After quantization, you're looking for a minimal increase in perplexity.

A more comprehensive quality check relies on a public intelligence benchmark or, better yet, a domain-specific eval that matches your expected real-world usage. On evals, you're looking for a minimal reduction in the quality score.

The best way to get the full picture of the impact of quantization is to run all three types of checks and make apples-to-apples comparisons to the original model weights.

Remember that quantization is a scale, not a binary decision. You can still get some performance improvements with lower risk of quality loss by quantizing to FP8 instead of FP4, or by quantizing fewer components of the model, like weights-only quantization.

If you're working in a highly sensitive domain and can't risk model quality, no worries: every other technique in this chapter is lossless in terms of quality.

5.2 Speculative Decoding

The decode phase of LLM inference is an autoregressive process in which tokens are generated one at a time. The bottleneck on decode is memory bandwidth, with compute sitting idle at low to moderate batch sizes as weights are read from memory.

Speculative decoding takes advantage of that spare compute to try to generate multiple tokens per forward pass through the target model. If an inference engine could generate two, three, or even more tokens for each round-trip of weights through memory, it would generate far more tokens per second. Speculative decoding only improves TPS/ITL, not TTFT.

There are multiple algorithms for speculative decoding. They share a common mechanism:

1. The speculator generates one or more **draft tokens**.
2. The **target model**, or the underlying model that you're trying to accelerate, performs **validation** on these tokens to check if they match what the model would have generated.
3. The target model accepts any valid draft tokens and generates an additional token itself, completing the forward pass.

This generates N+1 tokens per forward pass, or iteration through the decode loop, where N is the number of accepted draft tokens.

Generating draft tokens is not free, it takes both compute and memory. However, it is much faster for a target model to validate a draft token than to generate an original token. If you imagine a sudoku puzzle, solving it is hard, but checking if the solution is correct is very easy. For the target model, generating a token is like solving a sudoku, while validating a draft token is like checking a finished sudoku.

The performance uplift from any speculative decoding strategy depends on three factors:

1. **Draft token cost:** The time it takes to generate a draft token.
2. **Draft sequence length:** The number of draft tokens that are generated per forward pass.
3. **Token acceptance rate:** The percentage of draft tokens that are accepted by the target model.

Token acceptance rate is high early in the draft sequence, but draft tokens get less reliable deeper in the sequence.

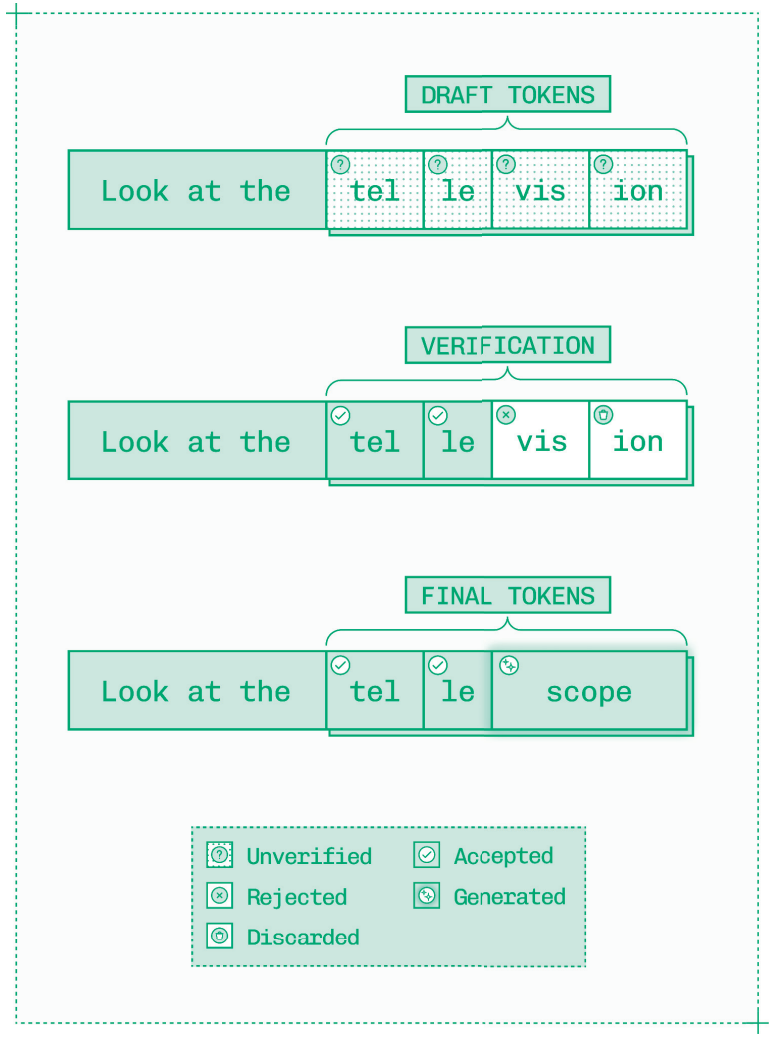


Figure 5.3: Speculative decoding from draft token generation and validation to prefix acceptance with subsequent token generation.

Aim for short, high-percentage sequences because generating and validating tokens, while inexpensive relative to generating tokens in the original model, still comes with meaningful overhead. Additionally, once a single draft token is rejected as wrong, all subsequent tokens in the sequence are also rejected.

Working with speculation is interesting because so many factors affect token acceptance rate. The big one is temperature – higher temperatures yield token distributions that are harder to predict, reducing the effectiveness of speculative decoding. But even factors as simple as subject matter can make a difference on acceptance rate if the draft model or additional head used for speculation is better versed in, say, math than history.

Another limitation on speculative decoding is that it's most useful at low batch sizes where there are spare compute cycles. At higher batch sizes, speculative decoding must be dynamically disabled as compute is too saturated to afford verification.

Each speculation algorithm navigates these tradeoffs differently, and careful implementation of the right algorithm for the situation can lead to major improvements in TPS.

5.2.1 Draft-Target Speculative Decoding

The original method of speculative decoding uses two models:

- **Draft model:** An additional model that generates speculative draft tokens.
- **Target model:** The original model, which now verifies draft tokens in addition to performing ordinary decode.

The most important decision when configuring draft-target speculative decoding is which draft model to use. A good draft model has a high token acceptance rate while requiring minimal resources to run.

Draft models are most often smaller members of the same family as the target model, as they share tokenizers and behaviors. As a rule of thumb, the draft model should be at least ten times smaller by parameter count than the target model. Fine-tuning or distillation can improve the token

acceptance rate of these small models by teaching them to behave more like the target model.

Draft-target speculative decoding is a good choice when you want something that is quick to set up out of the box without doing any training or fine-tuning.

However, other speculation algorithms generally offer better performance. Draft-target introduces the most overhead of any speculative decoding method. While the draft model is small, the inference engine must store the draft model's weights, activations, and KV cache in memory, and dedicate compute cycles to draft model prefill. Additionally, the draft and target model must run in coordination so that they don't compete for resources, though inference engines like TensorRT-LLM handle that model orchestration.

5.2.2 Medusa

Medusa was one of the first alternatives to draft-target speculation. Medusa addresses the complexity and overhead of operating a draft model to perform speculation by instead fine-tuning the target model to generate additional tokens per forward pass.

Fine-tuning a model for Medusa means grafting additional decoder heads onto the model. The ordinary architecture of an LLM contains a single decoder head, but Medusa adds an additional two to four heads that generate sequential draft tokens.

Like with draft-target speculation, draft tokens are validated on the next forward pass.

Medusa is still limited on draft token count and draft token acceptance rate, and it is not widely used in production today. However, Medusa inspired more popular techniques like EAGLE.

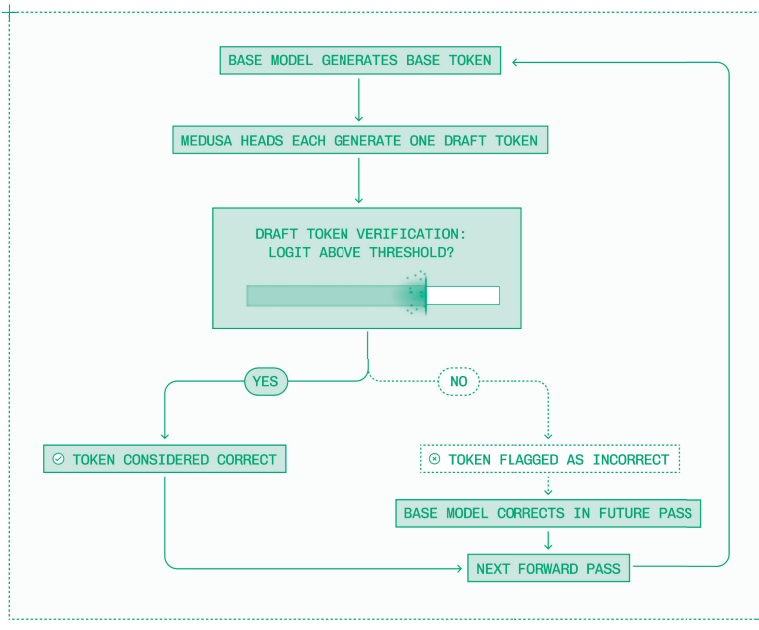


Figure 5.4: Medusa heads each generate a draft token on top of the token generated by the target model.

5.2.3 EAGLE

The main problem with using an off-the-shelf pretrained model as a draft model is that a model like Qwen 0.5B is designed to be a good standalone LLM on cheap hardware, not to speculate draft tokens on a B200. These draft models are inefficient to run and offer a relatively low acceptance rate.

EAGLE offers an alternative: a purpose-built draft model trained from scratch to generate sequences of up to eight draft tokens (two times more than Medusa) with a very high acceptance rate.

During inference, LLMs accumulate a lot of context about the predicted tokens in the form of hidden states between layers. Traditional draft models don't have access to this information.

EAGLE is a draft model trained to accept hidden states as input and generate speculative tokens as output. Specifically, it is trained on a set

of three hidden states: one from an early layer, one from a middle layer, and one from a late layer. EAGLE is often less than one billion parameters and scales well when given additional training data.

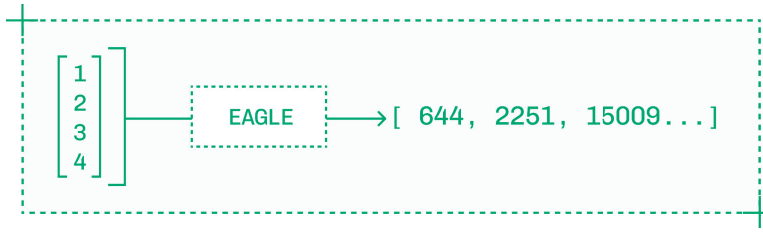


Figure 5.5: EAGLE speculative models take hidden states as input and produce draft tokens as output.

In practice, when using EAGLE with an inference engine like TensorRT-LLM, implementations tend to be straightforward, with post-training EAGLE creation and single-sequence speculation.

EAGLE can be attached to the same module (PyTorch class) as the target model, so each forward pass runs inference on both the target model and the EAGLE speculator. This unified pipeline solves the other problem with draft-target decoding, where multiple round trips to the CPU were needed to orchestrate the draft and target models.

EAGLE is the go-to speculation algorithm for general use among inference engineers with the knowledge and means to train EAGLE heads and is well-supported by inference engines. Like other speculation techniques, adopting EAGLE for improved latency requires reduced batch sizes, lowering throughput and increasing cost.

5.2.4 N-gram Speculation and Lookahead Decoding

N-gram speculation uses a different mechanism than other types of speculation. There is no draft model.

Instead, in parallel with generating the KV cache, the inference engine constructs an n-gram dictionary. The n-gram dictionary maps a single starting token to an observed sequence of N tokens (the n-gram).

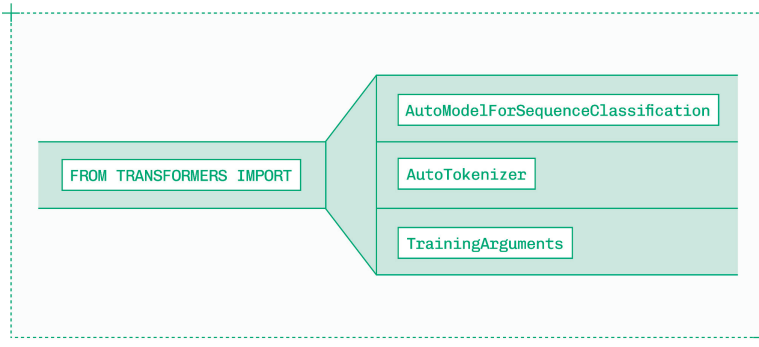


Figure 5.6: An n-gram dictionary matches prefixes to likely suffixes and is especially useful for constrained languages like code.

This n-gram dictionary contains common sequences from the input text and is first constructed during prefill. During decode, the generated token is fed into the dictionary, and any available suffix is selected as draft tokens. In the next forward pass, the target model verifies these draft tokens as normal.

The advantage of n-gram speculation versus EAGLE is that the sequences can be much longer. While EAGLE can generate eight or so draft tokens with a decent acceptance rate, n-gram sequences can exceed ten tokens.

However, the acceptance rate for n-grams is only high when the contents of the model output are similar to the model input. N-gram speculation is mostly used for code completion and code revision, where syntax is predictable and output closely matches input. However, within this specific domain, it easily outperforms EAGLE.

A similar method to n-gram speculation is Lookahead Decoding, which generates n-grams during inference to fill the dictionary. Lookahead Decoding is more general than n-gram speculation as it doesn't rely as much on highly repetitive context, but it requires extra compute to generate the n-grams.

Every speculative decoding algorithm aims to reduce the total number of forward passes needed to generate a complete output sequence, improving overall latency and specifically tokens per second per user during

decode. N-gram speculation excels at code completion and similar tasks, while Lookahead Decoding generalizes in systems with excess compute.

5.3 Caching

During prefill, the inference engine builds a KV cache (a store of keys and values for each token) on the input sequence. It then updates the KV cache for each token during decode. As inference is autoregressive, the value for each new token depends on the value of every previous token in the sequence.

Every inference engine uses KV caching by default on a request-by-request basis. Without KV caching, LLM inference would be unbearably slow as each previous value in the entire sequence would need to be re-calculated for each subsequent token.

However, engineers can get even more utility from the KV cache by re-using it between requests rather than solely within each inference sequence.

5.3.1 Prefix Caching and KV Cache Re-Use

Consider the following two prompts, each with four tokens on most tokenizers, in Figure 5.7.

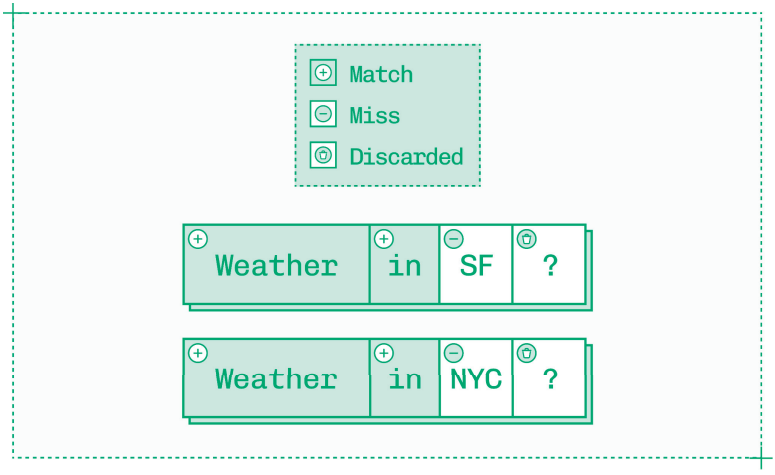


Figure 5.7: A pair of four-token sequences with two-token matching prefixes.

By default, the inference engine has to run prefill on all four tokens of each prompt. But the first tokens of each prompt – "Weather in" – form a shared prefix between the pair.

With prefix caching, you can re-use the KV cache from the first request to improve TTFT on the second request by skipping prefill on the first two tokens and reading in the existing KV cache instead.

When you see pay-per-token APIs charge less for "cache hit" input tokens than "cache miss" tokens, this is why – re-using cached tokens takes very little compute power or time. As an inference engineer, you can apply the same principle to reduce latency and improve throughput (thus saving money) on your own deployments.

Saving two tokens won't make a big impact on TTFT, but prefix caching can skip prefill on thousands of tokens in certain domains:

- **Complex system prompts:** Agents, customer-facing chatbots, RAG scaffolds, and tool calls often feature long, complex system prompts on every call.
- **Code completion:** Code completion, code generation, and other coding functions require passing the same thousands of lines of code as shared context.
- **Documents and retrieval:** Document summarization, question answering, and retrieval all add repeated context ahead of user prompts.
- **Multi-turn conversations:** Ordinary conversations repeat back every message in a chat template, increasing the savings from prefix caching with every turn.

Prefix caching works from the start of the input sequence until the first non-repeated token. The fourth token in the weather example, a question mark, is shared between the two input sequences. However, the prefix ends at the first non-repeated token, so the fourth token isn't read from cache.

Because prefixes end at the first unique token, your context engineering determines your TTFT savings. Consider a different approach to the same prompt:

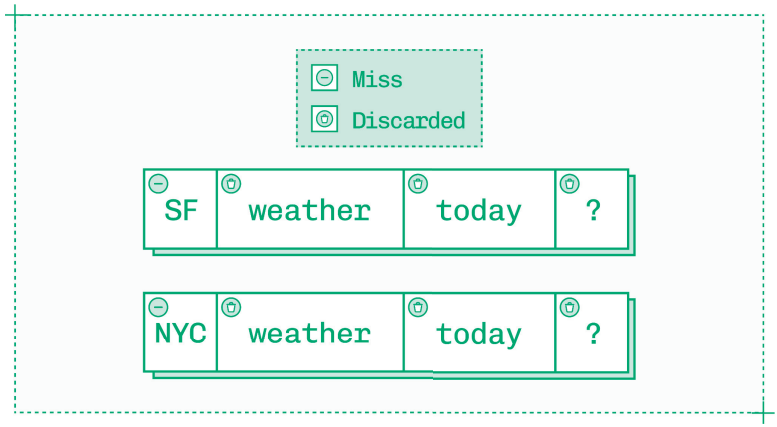


Figure 5.8: A pair of four-token sequences with no prefix match, the first tokens are different so it doesn't matter that the next three are the same.

Here, there is no savings from prefix caching as the very first token differs between the two sequences, even though every subsequent token is the same.

To take advantage of prefix caching, ensure that novel tokens are as late in your context as possible.

Prefix caching is the dominant form of KV cache re-use because LLMs are autoregressive. Each token influences every subsequent token, so a single novel token changes the way the model represents the rest of the sequence internally, even if the sequences look the same to a human reader.

However, there is active research around other kinds of KV cache re-use to overcome this limitation. Caching arbitrary sequences from the middle of prompts requires correcting both positional embeddings and selectively recomputing KV entries to maintain output quality. Tools like CacheBlend and LMCache support non-prefix sequences, expanding the possibilities for KV cache re-use.

5.3.2 Where to Store the KV Cache

The KV cache is very valuable. But KV caches take up a lot of memory, and GPUs only have limited VRAM.

You can configure how much memory your inference engine allocates to KV cache. For example, in TensorRT-LLM, you would set:

```
kv_cache_config.free_gpu_memory_fraction = 0.8
# or any value 0.0 < n < 1.0
```

Figure 5.9: Allocating free GPU memory to the KV cache is an essential configuration decision when running inference engines.

If you're working on a B200 GPU with 180 GB of VRAM and used 100 GB for model weights and buffers, this would allocate 80 percent of the remaining VRAM, or 64 GB, to KV cache.

Once this allocation fills – and it will fill quickly – you'll have to start deleting saved KV caches, increasing the chance of a cache miss on future requests.

To get more room for KV cache, offload from VRAM to other nearby storage. There are four places where you can store KV cache, in descending order of bandwidth to the GPU:

Level	Memory type	Approximate speed	Approximate size
G1	Device Memory (GPU VRAM)	Terabytes per second	10s to 100s of gigabytes
G2	Host Memory (CPU RAM)	10s to 100s of GB per second	100s of gigabytes to terabytes
G3	Local SSD	5-10 GB per second	Terabytes
G4	Networked SSD	Gigabytes per second	10s of terabytes

Certain SKUs, like the GB200, come equipped with CPUs and interconnects offering much faster G2 storage making them great for KV cache offloading.

NVIDIA Dynamo provides support for KV cache offloading via KVBM (KV Block Manager). KVBM provides APIs for moving KV cache blocks among different levels of memory. As a general rule, you want to keep the most frequently used blocks in higher-bandwidth memory, while less-often-used blocks can be relegated to slower storage until needed.

5.3.3 Cache-Aware Routing

In a production environment, there will be multiple replicas of your inference server, with incoming traffic split across the replicas. Usually, traffic is routed based on how busy each replica is.

If your inference server makes heavy use of prefix caching, your routing logic needs to be updated to account for that. A user in a long conversation with a chatbot or asking multiple questions about a codebase should have their request routed to the same replica whenever possible so that they get a cache hit for a faster, less expensive request.

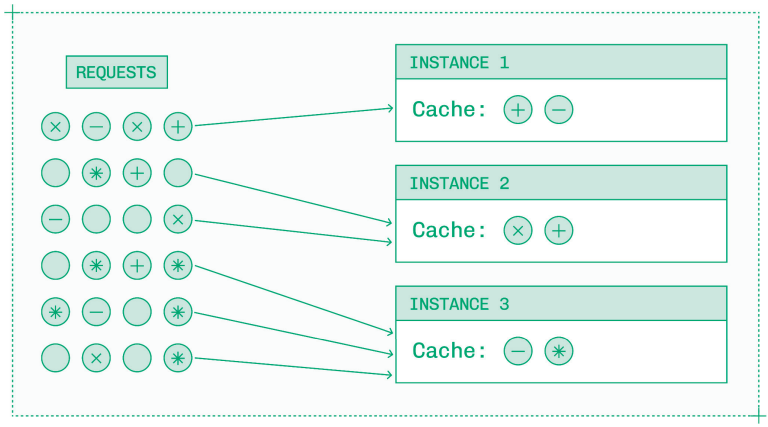


Figure 5.10: Cache-aware routing allocates traffic based on KV cache rather than simply dividing requests evenly across replicas.

Another option is using the G4 networked storage to build a global KV cache across replicas. Routing still matters here – a replica with a hot G1

cache will serve the request faster than a replica reading from G4 – but a global cache ensures that all replicas can eventually access any pre-computed sequence and that cached sequences are not lost when nodes cycle or are spun down during autoscaling.

5.3.4 Long Context Handling

"Long context" is a bit of a tautological definition: a sequence becomes "long context" when it generates a KV cache large enough to cause problems during inference.

Depending on the model, hardware, engine, and traffic, these problems can start to emerge past common cutoffs like 32K, 64K, or 128K tokens. In your performance benchmarking, be sure to send very large input sequences to test your inference service against long context requests.

Foundation model labs have been using scaling techniques like RoPE to unlock longer and more accurate context windows. But supporting these upgraded context windows introduces new challenges in inference.

Accounting for the KV cache, the attention equation scales linearly with sequence length. With long sequences, attention can become the main consumer of VRAM – the very resource decode is limited by.

While approaches like sliding window attention, compressed attention, and sparse attention offer solutions on a model-by-model basis, there are general approaches to optimizing the standard attention algorithm:

- **FlashAttention:** A series of optimized attention kernels to compute attention with reduced numbers of reads from and writes to memory.
- **PagedAttention:** A memory management technique that stores KV cache in fixed-size pages, reducing fragmentation and duplication.
- **Chunked Prefill:** A strategy of splitting large input sequences into chunks, which can be run alongside decode as resources allow to avoid overwhelming the inference engine with a long sequence.

But what if, after these optimizations, you still need more VRAM than a single GPU offers to store KV cache? You'll need to parallelize inference across multiple GPUs.

5.4 Model Parallelism

Every frontier LLM on the market today is too big to fit on a single GPU for batch inference. While GPUs have gotten bigger, so too have models, a trend that does not show signs of reversing.

In FP8, loading a billion parameters of model weights takes roughly a gigabyte of VRAM. For a model like DeepSeek-V3.1, with 671 billion parameters, the model weights alone would cause a single B200 GPU to immediately throw an out-of-memory (OOM) error.

It's not enough to just barely squeeze the model weights into VRAM. On 4xB200 GPUs, with 720 GB of VRAM, you could theoretically load DeepSeek's weights. But with no room left over for a KV cache, which often takes up 80 percent or more of the remaining VRAM after weights, four B200 GPUs would not be able to serve DeepSeek with any reasonable sequence length or batch size.

Instead, a full node of eight B200 GPUs is needed to serve real production traffic on a model the size of DeepSeek. You can estimate the minimum number of GPUs required for a model by multiplying the precision, parameter count, and expected KV cache allocation together.

```
# Deciding GPU allocation for DeepSeek

bits_precision = 8 # FP8 quantization
params = 671 # parameters in billions
kv_cache_allocation = 1.8

vram_required = (bits_precision / 8) * params * kv_cache_allocation

# DeepSeek requires about 1200 GB of VRAM with this config

b200_vram_available = [180, 360, 720, 1440]

round_up(vram_required, b200_vram_available)

# Serving DeepSeek in FP8 requires 1440 GB of VRAM (8xB200 GPUs)
```

Figure 5.11: After figuring out how much VRAM inference requires, round up to the next available instance size to determine minimum GPU count.

In many cases, even for midsize models like GPT OSS, you want to use more than the minimum number of GPUs required to enable larger KV caches and unlock better per-user latency.

However, all of this requires that inference scales efficiently from one GPU to multiple GPUs. The limitation in scaling parallel inference is the communication overhead between GPUs.

Chapter 3 details the different interconnects between GPUs: NVLink and NVSwitch within nodes, InfiniBand between nodes.

While NVLink and InfiniBand offer high bandwidth, they are a fraction of the speed of VRAM. With decode bound on memory bandwidth, multi-GPU inference needs to be carefully designed to avoid bottlenecks in inter-GPU communication. This field of study is called topology-aware parallelism.

There are three primary forms of model parallelism in inference:

- **Pipeline Parallelism (PP):** Splits the layers of the model across GPUs.
- **Tensor Parallelism (TP):** Splits the tensors within each layer across GPUs.
- **Expert Parallelism (EP):** Shards entire experts from MoE models across different GPUs.

Each form of parallelism has its own tradeoffs:

Method	Mechanism	Drawback
PP	Each GPU handles a stage of the forward and backward pass.	Not recommended due to poor latency and utilization from step-by-step pipeline.
TP	Compute-heavy operations like matmuls are split across GPUs.	Requires synchronization across GPUs, not suitable for multi-node.
EP	Each expert lives within a single GPU, making in-expert inference fast.	Requires routing between GPUs to reach multiple experts, better for throughput.

Tensor Parallelism is generally best for low-latency model inference within a single node, while Expert Parallelism improves throughput for MoE LLMs. Pipeline Parallelism is only used for multi-node inference.

Additionally, data parallelism strategies like Context Parallelism split computation across devices. These strategies are rare in LLM inference but essential for video generation (section 6.6).

5.4.1 Tensor Parallelism for Lower Latency

Tensor Parallelism should be your default strategy for multi-GPU model inference. It supports both dense models like Llama 405B and the MoE models that currently dominate the open model landscape.

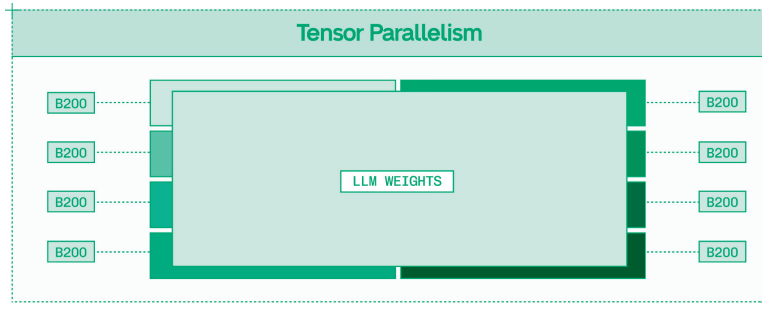


Figure 5.12: Tensor Parallelism splits weights across GPUs, effectively sharing VRAM resources to run large models fast.

TP works by splitting apart each layer of the model (as opposed to Pipeline Parallelism, which keeps layers intact) and distributing the layer fragments across the allocated GPUs. For each layer, the expense of reading from weights memory and executing matrix multiplication is shared across the GPUs.

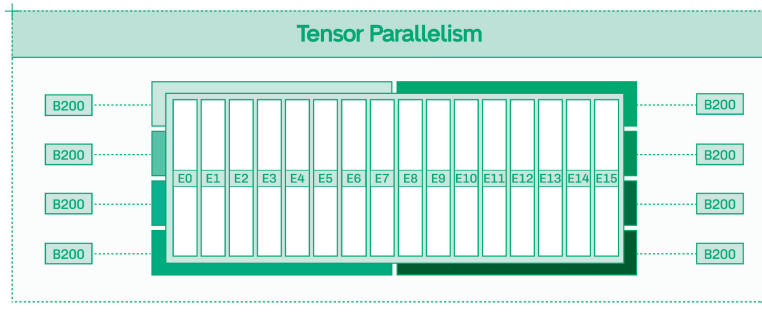


Figure 5.13: For Mixture of Experts models, each expert runs across multiple GPUs with Tensor Parallelism.

However, the results of each layer need to be communicated in an all-reduce fashion into a single output before the next layer can be computed. In nodes with high-bandwidth intra-node NVLink and NVSwitch, this communication overhead is minimized.

Increasing Tensor Parallelism improves TPS on a per-user basis (assuming the model is large enough and the sequences are long enough that the communication overhead doesn't outweigh the faster forward pass, which is the case for most frontier models).

5.4.2 Expert Parallelism for Higher Throughput

Expert Parallelism neatly divides experts across GPUs. In a model with 128 experts served in EP8 across eight GPUs, each GPU will host 16 full experts.

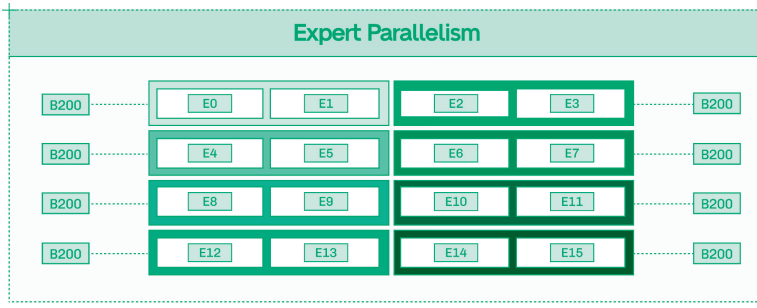


Figure 5.14: Expert Parallelism runs each expert within a single GPU, with GPUs each hosting multiple experts.

EP improves total system throughput, making inference more scalable and less expensive. With individual experts processing tokens separately, each token takes just as long, but the system as a whole can handle more simultaneous tokens.

Many deployments use a mix of TP and EP to achieve both benefits.

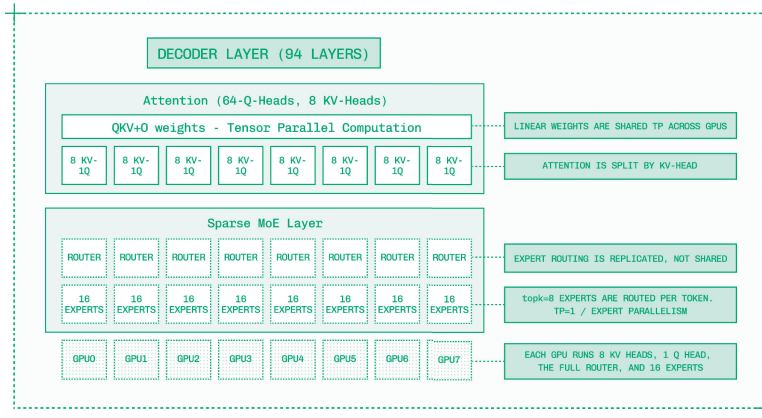


Figure 5.15: This deployment uses TP for attention and EP for the sparse MoE layer.

Expert Parallelism requires less inter-GPU communication than Tensor Parallelism. The Expert Router, which determines which experts each token activates, is replicated onto each GPU as it is a relatively small component of the model. Inter-GPU communication is necessary for passing tokens from expert to expert, but unlike TP, it is not required to collect the results of each layer.

Thanks to this lower communication overhead, EP scales well to multi-node deployments and systems with limited interconnect bandwidth.

5.4.3 Multi-Node Inference

If you're serving a huge model at high precision, supporting multi-million-token input sequences, or just trying to run inference as fast as possible, you might need more than eight GPUs.

GPUs are designed to work together across nodes, and multi-node training has been the standard for years to develop frontier models. But multi-node inference introduces new challenges:

- **Infrastructure:** How do you reliably provision two or more interconnected GPU nodes and build abstractions across cloud providers (chapter 7)?
- **Parallelism:** How do you effectively communicate over InfiniBand, which is much slower than NVLink?

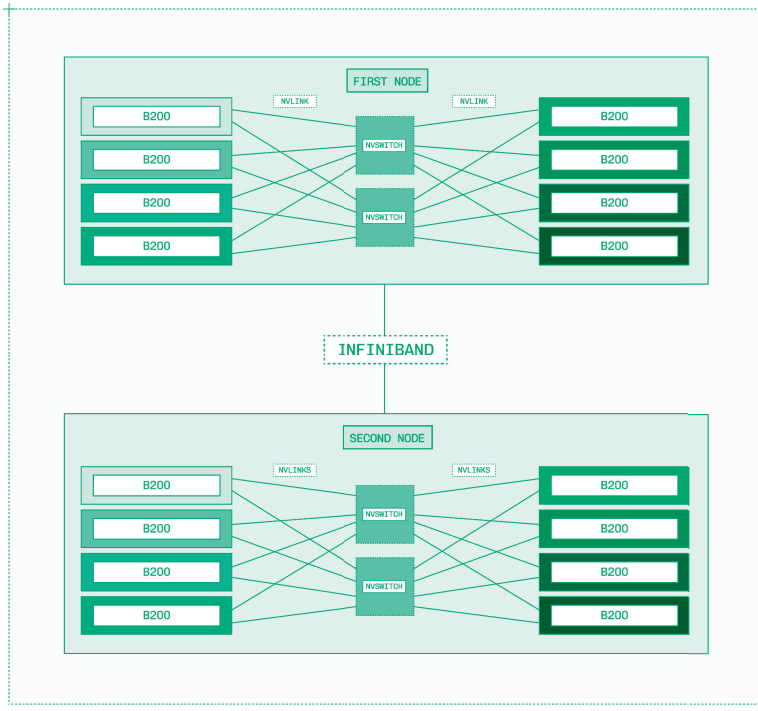


Figure 5.16: InfiniBand enables multi-node inference across more than eight GPUs via high-bandwidth node-to-node interconnect.

InfiniBand introduces a new wrinkle to topology-aware parallelism. Tensor Parallelism generally requires too much communication across GPUs to be a good fit for multi-node inference. Instead, you have two options that work well over InfiniBand:

1. For dense models, use Tensor Parallelism within each node and Pipeline Parallelism between nodes (e.g., TP8PP2).
2. For MoE models, you can also try Expert Parallelism (e.g., EP16) as it has a lower communication overhead than Tensor Parallelism.

For MoE models, TP8PP2 will generally offer lower latency per user and EP16 will yield higher overall system throughput.

Unless your model and KV cache are so large as to require multi-node inference, it probably isn't the best use of the extra hardware. You're often

better off using the extra nodes for horizontal scaling across replicas, or for disaggregated serving.

5.5 Disaggregation

Disaggregation combines three important ideas in inference engineering:

1. Prefill is a compute-bound process that determines your TTFT, while decode is a memory-bound process that determines your TPS.
2. Specialization improves performance in everything from kernel selection to inference engine parameter tuning.
3. You can effectively parallelize model serving over multiple GPUs, or even multiple nodes, if you can avoid bottlenecks from lower-bandwidth interconnects.

When prefill and decode run on the same node under heavy traffic, they have a higher chance of interfering with one another. Ideally, prefill uses more compute resources, while decode uses more memory, and the two can co-exist efficiently. However, with larger batches and more compute-intensive optimizations, prefill and decode start competing for resources.

5.5.1 How Disaggregation Works

Disaggregation, or disaggregated serving, is the idea of separating prefill and decode into separate engines on separate GPUs or nodes.

Disaggregation turns LLM inference into a three-step process:

1. The prefill engine takes the input sequence and generates a KV cache while computing the first token.
2. The prefill engine sends the KV cache over the hardware interconnect to the decode engine.
3. The decode engine computes all subsequent tokens.

In conditional disaggregation, the request is first sent to the decode engine, which checks if the input sequence is already cached or is short enough to handle locally:

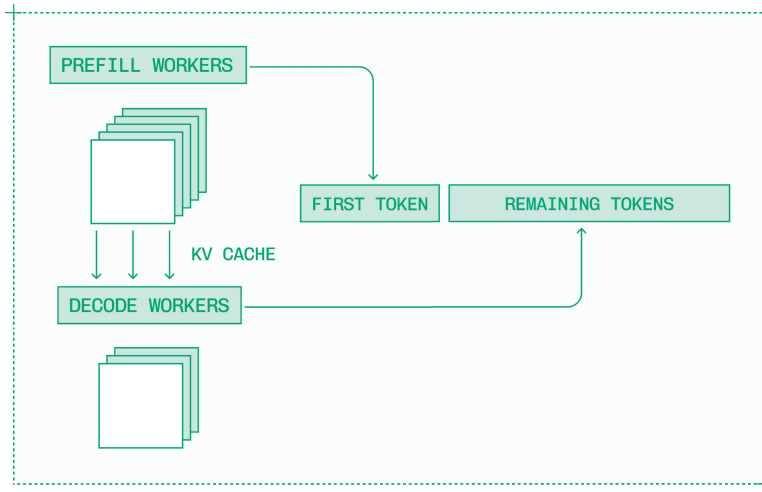


Figure 5.17: Disaggregation assigns prefill workers to generate the first token and decode workers to generate subsequent tokens.

1. If it is, the decode engine handles prefill locally, skipping disaggregation.
2. If it is not, the decode engine transfers the request to the prefill engine for disaggregated serving.

Conditional disaggregation is better for real-world traffic.

Another benefit of disaggregation is that with separate prefill and decode engines, you can optimize each engine individually and the system as a whole. For example, the compute-bound prefill engine requires a lower TP than the memory-bound decode engine.

5.5.2 When to Use Disaggregation

Disaggregation is very powerful but requires multiple GPUs and extra engineering work. You should reach for disaggregation only when:

1. You are serving a large volume of traffic, starting at one hundred million to one billion tokens per day depending on model size.
2. You are serving a larger model, at least a hundred billion parameters.
3. Your traffic is prefill-heavy with long input sequences.

If either point one or two is not true, you're likely wasting money on extra hardware for minimal performance gains. If point three is not true, you may be better off using the extra GPUs to scale replicas horizontally, as decode engines will be more efficient for short sequences or prefix cache hits.

A great use case for disaggregation is serving a frontier LLM in a code editor, where many developers are simultaneously passing in large and varied chunks of code as context. Tons of tokens, mostly prefill, on a trillion-parameter LLM is the textbook workload for disaggregation.

5.5.3 Dynamic Disaggregation with NVIDIA Dynamo

Dynamo provides production-ready support for disaggregation, with flexibility to handle heterogeneous real-world traffic.

Dynamo provides developer tools and pre-built optimizations to enable disaggregation:

- A prefill queue to hold requests when all prefill engines are saturated.
- Robust support for conditional disaggregation, with prefill routing based on configurable thresholds for ISL after prefix cache and prefill queue size.
- Efficient NIXL-based KV transfer from prefill to decode engines with a kernel to transpose KV blocks between layouts when the engines have different TP configurations.

Combined, these features enable dynamic disaggregation, where the number of prefill and decode engines is configurable at runtime and can be adjusted over time to match the changing nature of incoming traffic.

Disaggregation does not need to be a one-to-one ratio between prefill and decode engines. While it's simple to explain disaggregation in terms of a single prefill engine and a single decode engine, real systems have multiple of each.

The number of prefill and decode engines is written as xPyD, for example, 5P3D means five prefill and three decode engines working together to serve a single model deployment.

As systems grow more complicated, more potential bottlenecks appear. With disaggregation, the new bottleneck is prefill queue size. It's important to not let the queue grow too large, both by setting a reasonable threshold for local prefill on the decode engine and by reconfiguring xPyD at runtime to allocate more resources to prefill if needed.

The other potential bottleneck in disaggregation is running out of KV cache on the decode engines under high load. Increase KV cache availability with quantization and KV cache offloading.

CHAPTER 6

Modalities

Modalities

The modality of a model describes what types of input it accepts and what types of output it creates. Chapters 1 through 5 focus on inference engineering for LLMs, which take text as input and produce text as output. This chapter expands the discussion to more modalities.

Generative AI models offer a rich array of modalities, including:

Input	Output	Category
Text and image/video	Text	Vision language
Text or image/video	Vector	Embedding
Audio (voice)	Text	Transcription
Text	Audio (voice)	Speech synthesis
Text	Audio (music)	Music generation
Audio (voice)	Audio (voice)	Speech-to-speech
Text and/or image	3D model	Generative CAD
Text and/or image	Image/Video	Image/video generation
Image/video	Text	Captioning
Image/video	Mask	Segmentation
Text and image	Image	Image editing

Fortunately, while there are many modalities, there are just two broad archetypes of generative AI models as outlined in chapter 2:

- **Autoregressive token generation:** Start from a tokenized sequence and predict the most likely next token.
- **Iterative denoising:** Start from random noise and refine toward the most likely output.

LLMs are the most famous autoregressive transformers models for token generation, but far from the only ones. Vision language models, text and multimedia embedding models, automatic speech recognition (ASR) models, text-to-speech (TTS) models, and many others rely on similar architectures.

Many of the same inference engines and techniques used for LLMs also apply to these related modalities.

Image and video generation models instead rely on iterative denoising, though increasingly hybrid diffusion transformer models are setting the frontier in quality. While a number of the same philosophies from kernel selection to parameter tuning also apply to image model optimization, the details end up quite different.

For each new modality, you also need to adjust the way you think about and measure latency, throughput, and quality. For example, a single token of audio output from a TTS model isn't particularly useful; instead of TTFT, measure the time to first word or time to first sentence.

This chapter discusses inference engineering for six common modalities beyond LLMs, with special attention to the different considerations for each modality.

6.1 Vision Language Models

Vision language models (VLMs) take one or more images or videos as input along with a text prompt and generate a text response.

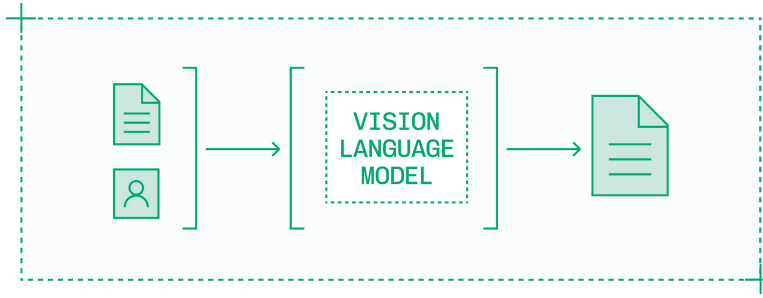


Figure 6.1: VLMs add image and video understanding to LLMs.

A vision language model usually consists of two modules:

- **LLM:** A standard large language model.
- **Vision encoder:** A small model that takes raw images and videos as input and converts them into image tokens.

The language model is much larger than the vision encoder. For example, in Mistral Large 3, the vision encoder is just two billion parameters compared to the 673B-parameter LLM.

While the vision encoder is small by parameter count, it is critical for inference. VLMs use varied architectures and implementations for vision encoders, so runtime support is somewhat more fragmented for vision language models. This fragmentation increases the importance of vLLM and SGLang for serving vision language models.

As a rule of thumb, sending a high-resolution input image to a VLM adds about a thousand visual tokens to the input sequence. While at a very high level image tokens are similar to regular tokens, they add up quickly.

Across VLMs, the primary challenge in inference optimization is handling the longer input sequence and larger KV cache. This adds wrinkles at both phases of inference:

- **Prefill:** Images are patched, embedded, tokenized, and fed into prefill as part of the input sequence.
- **Decode:** Same mechanics, longer context, and some models add attention variants for the image tokens.

Every technique from the previous chapter is useful in addressing this challenge:

- **Quantization:** KV cache quantization reduces the memory bandwidth and storage overhead for longer sequences.
- **Speculation:** Decode for VLMs matches LLMs and can be accelerated with speculation, especially EAGLE.
- **Prefix caching:** Re-use KV cache for images in multi-turn chats and repeated queries.
- **Parallelism:** Use Tensor Parallelism for fast inference while accessing more VRAM for large models and long contexts.
- **Disaggregation:** Move prefill to specialized and independently scaling workers to handle long sequences.

In addition to these techniques, VLMs introduce a new quality-speed tradeoff: downsampling. Images and videos can be converted into visual tokens at various resolutions. A high-resolution representation takes about four times more tokens than a low-resolution image, but provides more detailed information. Downsampling generally isn't needed for single-image inputs, but it may be needed when passing in multiple images or video clips.

6.1.1 Video Processing for Vision Language Models

A video is more than the sum of its frames. Videos may contain audio (though many VLMs cannot process audio, which must be transcribed separately and added into the prompt) and their frames express motion of objects through space that is lost when looking at static images.

VLMs are trained on video clips to understand that time dimension. High-quality inference requires processing the entire video clip in a single call to the model.

One second of cinematic video contains 24 frames. Each frame is an image. If a high-definition input image takes about 1,000 tokens to represent, then a four-second video clip produces an input sequence of nearly 100,000 tokens.

In reality, video inputs don't generate quite this long of an input sequence – downsampling is practically obligatory.

Reducing the resolution and frame rate makes it possible to evaluate an entire clip in a single inference request, though video understanding models are still only capable of taking very short clips.

After the video is tokenized and encoded, inference is similar to working with images, just with much longer context. Prefix caching, KV cache offloading, and optimized attention implementations are of even greater importance for these input sequences of tens of thousands of tokens.

6.1.2 Omni-Modal Models

Vision language models are an important part of a trend toward "omni" models that accept multiple types of input and produce multiple types of output. There are pros and cons to omni models – their blend of modalities provides unique capabilities, but smaller specialized models are often faster and more accurate within specific domains.

For example, many VLMs have text recognition capabilities trained into their image input processing. However, these capabilities lag behind dedicated optical character recognition (OCR) models that are generally a fraction of the size.

Running production inference on VLMs often involves coordinating a pipeline of multiple models and pre-processing steps. You might have individual preprocessors for extracting data from PDFs, reading text from images via OCR, or transcribing audio from a video.

Each component in the pipeline must be individually optimized for speed and should scale independently to avoid bottlenecks.

6.2 Embedding Models

An embedding model transforms a variable-length chunk of text – or another modality of input like an image – into a fixed-length vector representation that captures the semantic meaning of the input.

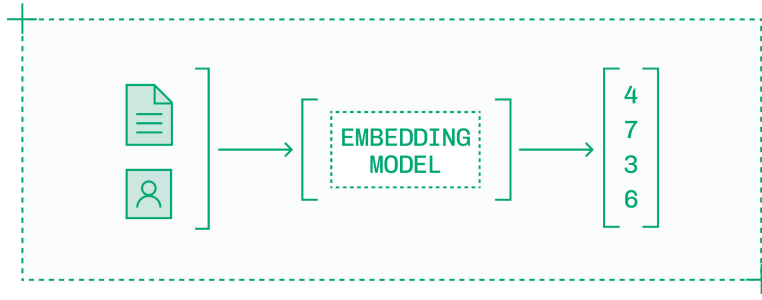


Figure 6.2: Embedding models convert unstructured input data into vectors that encode semantic meaning.

By encoding content into this shared semantic vector space, you can compare distance between items with simple math. Embedding models (along with vector databases) are used to build agent memory, RAG, search, and recommendation systems.

To support these use cases, embedding model inference workloads have two different traffic profiles:

1. **High-throughput backfills:** Bulk operations like indexing millions of documents, updating product catalogs, or even preparing data for LLM pre-training.
2. **Low-latency lookups:** Individual user-facing queries for search, retrieval, or recommendation, where every millisecond affects user experience.

Inference engineering for embedding models starts with clarifying which profile you need to serve. If you need to do both and have enough traffic to justify the cost, it's better to build a separate system for each type of usage.

6.2.1 Embedding Model Architecture

There are tens of thousands of embedding models on Hugging Face, but they all use one of two transformers-based architectures:

- **BERT-style models:** Encoder-only neural networks, usually <1B parameters, originally built for masked token prediction.
- **LLM-based models:** Modern language models, generally <=8B parameters, repurposed to generate embeddings.

Today, LLM-based embedding models offer substantially greater capabilities, though BERT-style models are still used for simple latency-sensitive tasks like classification.

Embedding models introduce their own speed/quality tradeoff in embedding dimensionality, or the size of their output vectors. An embedding vector contains a few hundred to a few thousand values, with longer vectors encoding more information.

Modern embedding models use Matryoshka representations to unlock dynamic tradeoffs between embedding dimensionality and quality while retaining more information on shorter vectors. Dimensionality doesn't materially affect inference time but does affect the storage, retrieval, and similarity computation time within a system.

In most cases, vectors from one embedding model cannot be meaningfully compared to vectors from another embedding model, even if they are the same length, as they encode inputs into different semantic spaces.

6.2.2 Embedding Model Inference

For embedding models with LLM backbones, like Qwen 3 Embed 8B, inference optimization shares common tools and techniques with other high-volume, low-latency deployments of smaller LLMs.

There are multiple runtimes for text embedding models: vLLM, SGLang, Infinity, TEI (Text Embedding Inference by Hugging Face). But the best performance comes from adapting TensorRT-LLM to run these models.

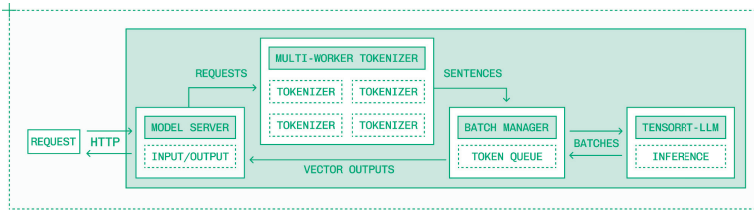


Figure 6.3: A high-performance embedding inference pipeline adds parallel tokenization and batch management in front of an optimized inference engine.

TensorRT-LLM brings an optimized XQA kernel for fast attention and kernel fusion techniques to reduce memory access overhead. For supported models, TensorRT-LLM is the most performant inference engine for both latency and throughput.

Further gains come from quantization. While smaller models are more likely to lose quality from quantization, FP8 quantization for embedding model weights offers improved performance with minimal quality loss.

The easiest way to check embedding model quality post-quantization is to run the same inputs through both the original and the quantized model, then check the cosine similarity of the output vectors. A cosine similarity of one hundred percent means the vectors are identical; you'll want to see at least 99 percent similarity to have confidence in the quantization.

As embedding models process tokens in parallel, prefix caching and disaggregation aren't relevant optimizations. And given these models' small size, parallelism across multiple GPUs is not effective. Instead, high-traffic deployments should scale horizontally, with each GPU as its own replica.

In high-traffic deployments of embedding models, batching and queueing play an important role in performance. Embedding models offer much higher batch sizes than other models. A single request may batch dozens or hundreds of text inputs together in a list, and many requests can run in parallel on a single GPU as even the most demanding embedding models are relatively small and fast.

Whether you're performing a large backfill or handling a surge in usage, traffic can exceed even the large batch sizes offered by embedding models. In these cases, a robust queuing system is essential infrastructure for supporting embedding model inference.

6.3 ASR Models

Automatic speech recognition (ASR) models take audio as input and produce text as output, powering transcription and dictation apps. The most popular open ASR model is Whisper, which was released by OpenAI. Whisper supports dozens of languages with accurate transcription.

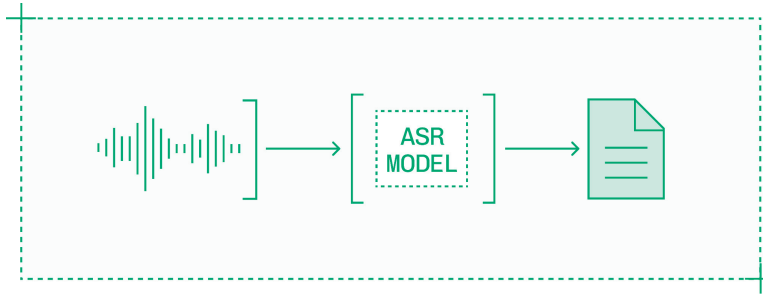


Figure 6.4: ASR models transcribe input audio into text.

Whisper comes in various sizes, but the largest and highest-quality Whisper model is just 1.55B parameters. Whisper runs extremely fast on fractions of large GPUs like H100 via Multi-Instance GPUs (MIGs). While various other sizes, variants, distillations, and quantizations exist, in practice it's possible to satisfy most latency budgets with the highest-quality models: Whisper 3 Large and Whisper 3 Turbo.

Whisper is an encoder-decoder model:

- **Encoder:** Takes a processed audio waveform (log-Mel spectrogram) as input and encodes it into audio features.
- **Decoder:** Takes these encoded audio features and converts them into text tokens.

The overwhelming majority of inference time is spent on the decoder, which is an autoregressive transformer model very similar in architecture to an LLM. Fortunately, there are excellent tools for optimizing the main bottleneck.

The main tool for performance optimization on the decoder side is TensorRT-LLM. With TensorRT-LLM, you can get in-flight batching for the decoder and an optimized C++ runtime with highly efficient CUDA kernels. TensorRT-LLM works especially well with recent architectures like Hopper and Blackwell, making MIGs an even better option for ASR inference.

6.3.1 Single-Chunk Latency Optimization

One use case for Whisper is real-time transcription, like in a dictation app or voice agent.

For live Whisper, look at round-trip time for a single chunk of audio to be transcribed. A great target to aim for is 200 milliseconds, which is the average human reaction time.

With Whisper running on an optimized TensorRT-LLM inference engine, there isn't a lot of work to do on the runtime level to improve performance. Instead, most gains for real-time Whisper come from orchestration and infrastructure.

The biggest upgrade in product experience for ASR is streaming, which is implemented at the API server layer rather than the model runtime layer. By establishing a WebSocket connection (section 7.5.3) and streaming audio continuously in and text continuously out, products can transcribe in real time rather than transcribing pre-recorded audio.

At the ASR runtime layer, nothing changes. Instead, a streaming implementation for transcription uses a Voice Activity Detection (VAD) model to monitor the incoming stream and segment it into discrete chunks for the ASR model to process. Inference is run on the chunks as normal, and the text results are streamed back via the WebSocket.

This setup can handle several concurrent streams, and it has the advantage of keeping transcription sequential. When each chunk is processed on the same GPU, you can use the output sequence of the previous chunk as the prefix for the next chunk, improving transcription quality.

6.3.2 Long File Latency Optimization

One limitation of the Whisper model is that it can only support 30-second chunks. Transcribing long files, like hour-long podcasts, requires a different set of optimizations.

Measure the performance of long file transcription with the confusingly named Real-Time Factor (RTF). If the world's fastest typist could manually transcribe an hour of audio in 30 minutes, they would have an RTF of 2X. With a Whisper deployment optimized for long files, you can transcribe an hour of audio in less than four seconds, for an RTF of 1000X.

Fast transcription for long files requires a multi-step pipeline. The first step is again a VAD model, this time running on its own dedicated hardware. The model is used to remove silence and chunk out meaningful audio segments rather than splitting by time intervals, which runs the risk of cutting words in half.

Then, the chunks can be processed in parallel. Ideally, you use multiple GPUs (or multiple MIGs) to process more audio chunks at once. RTF improves roughly linearly with the number of GPUs used. Each GPU processes multiple chunks at once with in-flight batching for high utilization.

Finally, the chunked transcripts are stitched back together by timestamp.

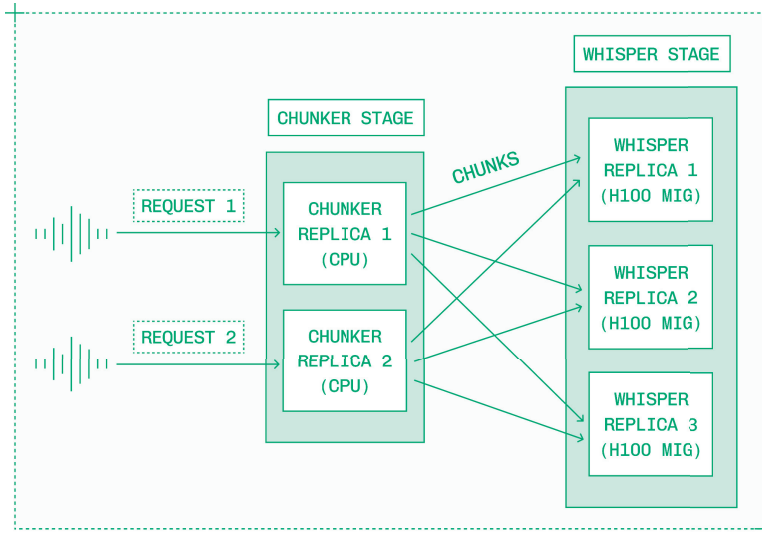


Figure 6.5: A two-stage pipeline for long audio file transcription parallelizes chunk transcription to improve end-to-end request time.

Parallelizing chunk transcription removes the ability to use the previous sequence as a prefix for the next sequence. But there are other quality improvement techniques that more than make up for this.

With ASR output, you can automatically detect hallucinations like repeated words and phrases by measuring the compression ratio and words per minute of the output. When a chunk has an issue, you can:

1. Re-run the chunk with a higher temperature. This is counterintuitive – higher temperatures generally produce more hallucinations – but the intention is to break cycles of repeated words and generate a different output.
2. Re-chunk the entire audio, or a segment of the audio, into smaller chunks and re-run the transcription.

In practice, these techniques obviate the need for passing a previous sequence as a prefix, unlocking highly efficient and accurate parallel transcription for long files.

6.3.3 Diarization

Diarization, or annotating a transcript with who is speaking when, is an adjacent problem to transcription. Diarization models categorize audio by voice feature, then segment and cluster across the file to timestamp changes in speaker.

Diarization models are a completely different class of model. Where Whisper is an encoder-decoder transformers model, diarization systems like pyannote audio are pipelines of classic ML models.

A diarization pipeline contains models for segmentation, embedding, and clustering. To optimize diarization, you have to run each model fast and orchestrate the whole pipeline efficiently.

As diarization is an ML pipeline, you can use tools like PyTorch and pyannote along with optimizations like Torch compilation to improve its performance. In practice, even highly optimized implementations of diarization take at least twice as long to process an audio file versus transcription.

6.4 TTS Models

Text-to-speech (TTS) models, also called speech synthesis models, take text as input and produce audio as output, specifically generating speech. In 2025, open models like Orpheus TTS introduced extremely lifelike speech synthesis to the open model ecosystem. Many companies fine-tuned Orpheus for increased vocal quality and product-specific voices, leading to high adoption of open models in the voice AI space.

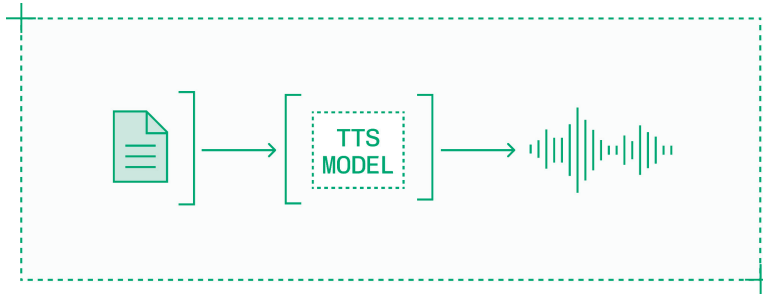


Figure 6.6: Text to speech models synthesize input text into audio.

Modern TTS models are fine-tuned LLMs. Orpheus TTS, for example, is derived from Llama 3.2 3B. This means that many of the same runtime and performance optimizations developed for LLMs apply to speech synthesis models.

TTS models have a small parameter count – Orpheus TTS at three billion is on the larger end – meaning that like ASR models, MIGs on H100s are highly efficient and performant options for inference.

Unlike ASR models, which generally run in FP16, TTS model weights and KV cache can be quantized to FP8 for better performance in addition to the optimized kernels and in-flight batching introduced by the TensorRT-LLM inference engine.

TTS models with LLM backbones are trained by expanding the vocabulary size of the LLM with tens of thousands of encoded audio tokens. Then, the models are trained on pairs of text inputs with tokenized audio outputs. This means that to use TTS models in practice, you also need an audio decoder that takes the audio output tokens and converts them into a waveform.

This audio decoding process adds a potential bottleneck to inference. The audio decoder should be implemented using PyTorch and compiled for efficient operation on the target GPU and should use dynamic batching with a short timeout (e.g., 15 milliseconds). In-flight batching is not possible for the audio decoder.

TTS model performance is measured with somewhat different metrics than LLMs. The key metrics are:

- **TTFB:** Time to first byte (TTFB) is the equivalent of TTFT for speech synthesis.
- **Time to first sentence:** Instead of TTFB, a more user-oriented latency metric is the time to generate the first meaningful phrase or sentence.
- **TPS:** Like an LLM, the TTS model generates tokens, so decode speed can be measured in tokens per second.

Like with TTFT on LLMs, the goal is to minimize TTFB for speech synthesis. For Orpheus, it's possible to get as low as 150 milliseconds on a single H100.

However, there are different goals for TPS on speech synthesis models. The tokens that the model generates are converted to audio waveforms. Depending on the model, it might take 80 to 100 tokens per second to generate audio in real time. Beyond that level, there isn't any benefit to generating additional tokens per second.

Instead, performance enhancements are used to scale throughput in terms of the number of concurrent real-time outputs the model can create. If a single GPU can support many concurrent users, the per-user cost of speech synthesis drops dramatically.

6.4.1 Streaming Real-Time Text to Speech

Most TTS tasks call for real-time speech synthesis. Like with ASR models, the performance gains for real-time systems come less from the runtime layer – which has already been optimized with TensorRT-LLM, quantization, and a compiled SNAC decoder – and instead from infrastructure.

Again, streaming over WebSockets is the biggest unlock for performance versus sending text and receiving audio in discrete chunks. After testing the inference engine to determine how many concurrent real-time streams can be generated, set the same batch size and active WebSocket count to keep usage high but stable.

TTS models are rarely used outside of real-time applications. However, if you do end up with a batch use case like backfilling a large corpus of documents to audio for improved accessibility, note that TTS models don't do well with long inputs, speech starts to degrade after 30 seconds or so.

6.4.2 Speech-to-Speech Models

One exciting area of research is speech-to-speech models, or models that take audio as input and generate audio as output.

Today, most voice systems use a cascading approach, where an ASR model, LLM, and TTS model work in a pipeline to listen, think, and respond to users. These pipelines also employ auxiliary components like VAD and embedding models to facilitate natural conversation and add context.

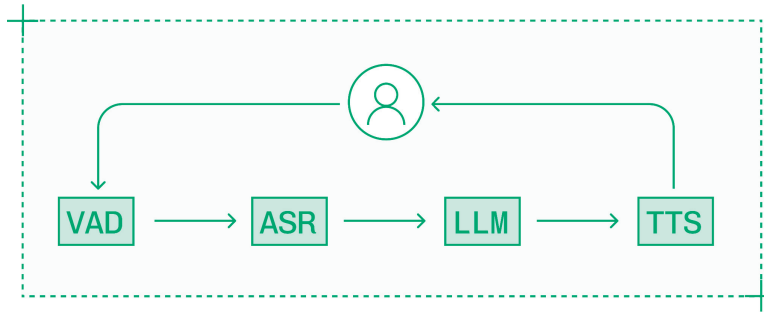


Figure 6.7: Most voice-based applications use a cascading approach with a multi-model pipeline.

Speech-to-speech models, like OpenAI's gpt-realtime, augment a core LLM with audio consumption and production capabilities, effectively unifying the pipeline in a single model. This is possible thanks to ASR, LLM, and TTS sharing such similar architectures, especially on the decoder.

At the time of publication, there are no commercially viable open speech-to-speech models, and closed options like gpt-realtime are significantly less capable and more expensive than cascading multi-model setups. However, research in this space is robust, and this emerging modality will soon require its own flavor of inference engineering.

6.5 Image Generation Models

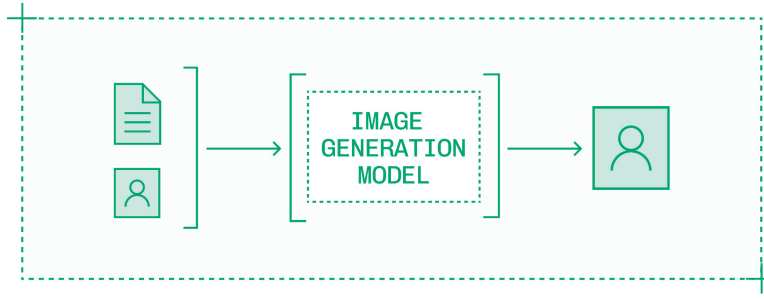


Figure 6.8: Image generation models may accept both text and reference images to create new output images.

Working with image and video generation models is entirely different from working with large language models on a few axes.

The first is architecture. While some recent models like HunyuanImage-3.0 more closely resemble LLMs, most image and video generation models are iterative denoisers, not autoregressive token generators. Image generation models are pipelines with multiple small models working together in latent space rather than the uniform decoder architecture of an LLM.

As such, the tooling is different. At the time of publication, SGLang Diffusion and vLLM Omni are brand new. Most image and video generation model inference is implemented lower in the stack, working with PyTorch or TensorRT directly.

The constraints are different too. Image generation models are ten to twenty times smaller than frontier language models, and inference is constrained on compute, not bandwidth.

But perhaps the most significant difference is that image and video generation models offer more direct quality to speed tradeoffs.

Evaluating image model output quality is difficult to do programmatically. Automatic pipelines using vision language models give directional signal at best and may diverge from human preferences. The human eye is mysterious, and most image quality evals work by asking humans to pick among thousands of images to aggregate vibes and preferences into quality benchmarks.

6.5.1 Image Generation Kernel Optimization

When you read a model card for an image generation model from its repository, inference examples generally use the `diffusers` library with very few optimizations.

In fact, while image generation is theoretically compute bound, you often need to select memory-efficient kernels and use kernel fusion to even reach that bottleneck.

High-performance image model inference uses one of three libraries:

- **SGLang Diffusion:** Performant inference engines built for popular image and video generation architectures.
- **TensorRT:** High-quality black-box implementations of popular models with NVIDIA's in-house kernels.
- **PyTorch:** Careful kernel selection and fusion yields control, flexibility, and improved high-end performance.

If you want something that works well and you want it now, just use the SGLang Diffusion or TensorRT implementation of a model. But with PyTorch, there's an opportunity for advanced inference engineers to do deep customization.

The most essential kernel is the attention kernel. Many image generation models use FlashAttention 2 out of the box, but FlashAttention 3 and 4 yield better performance on Hopper and Blackwell GPUs, respectively.

There is a whole barrage of smaller kernels, especially normalization functions like RMSNorm, that are good candidates for fusion to ensure efficient memory usage.

Then, GEMM kernels matter for compute-bound inference. GEMM kernels apply to linear layers, and are generally safe to quantize into 8-bit floating point formats to access two times higher FLOPS on Tensor Cores. Kernels from CuTe, CUTLASS, or DeepGEMM may prove most efficient on a model-by-model basis.

Torch compilation includes automatic kernel fusion with a plugin system for inserting manually selected kernels, and the resulting engine can be cached for faster load times on node startup (which is important because compilation takes several minutes).

Like most high-performance engines, Torch compilation targets the specific GPU model and architecture performing the compilation – if you want to run the model on a B200, do the compilation on a B200.

6.5.2 One Weird Trick for Faster Image Generation

Kernel selection and Torch compilation are all bona fide inference optimization techniques. But the world of inference optimization has fun hacks as well, and here's one of them.

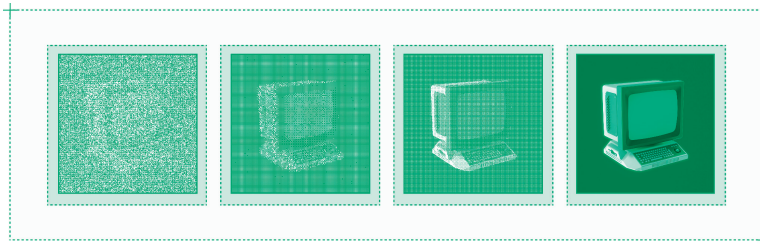


Figure 6.9: Recall that diffusion is a step-by-step process and that the general outline of the image is established in early steps.

Image generation time tracks linearly with step count. That's why few-step models and latent consistency models are so much faster than full 50-step models. But reducing step count may reduce image quality below an acceptable threshold.

Each pass through the denoising model is run at a batch size of two as each step includes a pass with and without prompt guidance.

As a refresher, the guidance parameter controls how much the prompt-guided image is weighted when combining the two iterations generated on each step. If the guidance is zero, the prompt-guided image does not need to be generated.

After the first few steps, the basic outline of the image is in place, and the rest of the steps are for filling in the details. Thus, prompt adherence is more important in early steps that affect the broad strokes of the image – the model is not going to change its mind on later steps and generate a dog when it is in the middle of generating a cat.

If you turn off guidance partway through the image generation, you save passes through the denoiser without reducing step count. If guidance is skipped for the last 20 steps of a 50-step run, there are only 80 passes through the model instead of 100, and quality generally remains high.

6.6 Video Generation Models

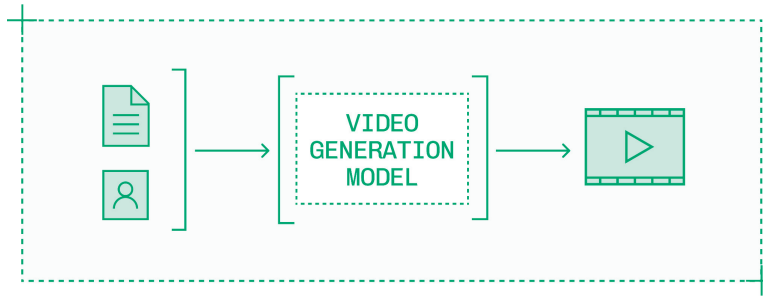


Figure 6.10: Video generation models take text prompts and may take keyframes or other image, audio, and video input.

Video generation is the most demanding modality. Whenever possible, these models should be run on Blackwell GPUs (or Rubin, once available). These GPUs offer a high memory capacity for Context Parallelism, fast Tensor Cores for attention computation, and microscaling data formats for more precise quantization.

Architecturally, video generation is similar to image generation, just rendering a full video rather than a single frame from latent space. Following the principle that greater scale unlocks more techniques, video generation uses all the same techniques as image generation plus additional optimizations.

Like image generation, video generation is compute bound and works via iterative denoising over latent space. Video generation models generally take about the same number of denoising steps as image generation models (~50), but each step processes much more data.

As video generation models are compute bound, batching isn't useful like it is for text generation. Video generation models usually run on full nodes of eight GPUs with a batch size of one: all eight GPUs work together to create one video at a time.

Unlike with batched workloads, where latency-throughput tradeoffs are possible by adjusting the batch size, the only way to improve the throughput and cost of video generation is to make the model itself faster.

Early video generation models were framewise. They generated frames one at a time. This reduced the quality and coherence of the video output. Today, video generation models run denoising steps on the video as a whole in latent space. Where latent space for image generation represents two dimensions (width, height), for video models it represents three (width, height, time).

This means passing huge amounts of data through each attention calculation. For video models, attention is 70 to 80 percent of the compute time, making attention the most important thing to optimize.

6.6.1 Attention Optimization and Quantization

Attention optimization starts with kernel selection. Test FlashAttention, DeepGemm, CuTe, and CUTLASS kernels to see which ones perform best for your model.

Where language models use the KV cache to accelerate attention, video generation models use other caching patterns to attempt to reuse model outputs. Re-using parts of the attention computation can make video generation 30 to 40 percent faster in practice.

Precise methods and algorithms are continuously changing with new research, but there are two fundamental approaches to caching:

- **Timestep-based caching:** Caching and re-using the outputs of certain timesteps to skip entire steps.
- **Transformer-based caching:** Caching and re-using hidden states to skip layers within the transformer itself.

Algorithms and implementations range from negligible quality degradation to unusable output – test these strategies carefully before using them in production.

Beyond kernels and caching, the main tool for speeding up attention is quantization.

For bandwidth-constrained language model inference, the benefit of quantization is that it means you have less data to load through memory. For

video models, it means you access double the FLOPS by switching to lower-precision Tensor Cores.

However, language model quantization focuses on weights – large linear layers where the impact of quantization is negligible. For video models, quantizing weights still helps, but while these layers take the majority of the memory bandwidth, constraining language models, they're only a small fraction of the compute time for video models.

Instead, quantization on video models focuses on attention. Attention is the riskiest part of any model to quantize, as errors accumulate over the course of inference. For video models, where there are ~50 steps instead of the thousands of autoregressive iterations in token generation, the risk is slightly lower but still important.

The first method for reducing the quality impact is to use a blockwise quantization and a microscaling data format (MXFP8), both available on Hopper and Blackwell. Microscaling data formats do a better job of preserving outlier values, which have a major impact on attention accuracy.

The most sophisticated approach to attention quantization is selectively quantizing within the model by:

- **Step:** Keep early steps in FP16 and quantize later steps.
- **Layer:** Keep first and last layers and quantize hidden layers.

Quantization by step follows the same insight as the classifier-free guidance trick from image generation models: early steps establish the outline of the image, while later steps refine the details. These early steps are more important for prompt adherence and accuracy.

For layers, the first and last layers are more important as they take the input and produce the final output. The hidden layers only perform intermediate calculations which don't suffer as much from approximation.

By only quantizing less important parts of the video generation process, quality is preserved. These tactics are found in kernels like SageAttention, an 8-bit attention kernel that you can use for quality low-precision attention on video generation models.

6.6.2 Context Parallelism

While video generation models generally run on a full node of eight GPUs, they use Context Parallelism rather than Tensor Parallelism.

Context Parallelism copies the weights onto every GPU. Video models are small enough that replicating the weights eight times takes a meaningful amount of memory but is feasible on B200.

Instead of splitting the model across GPUs, Context Parallelism works by splitting the attention calculation across the GPUs. This is coordinated via a mechanism like ring attention, where each GPU holds a piece of the context and passes intermediate results to the next GPU in the ring.

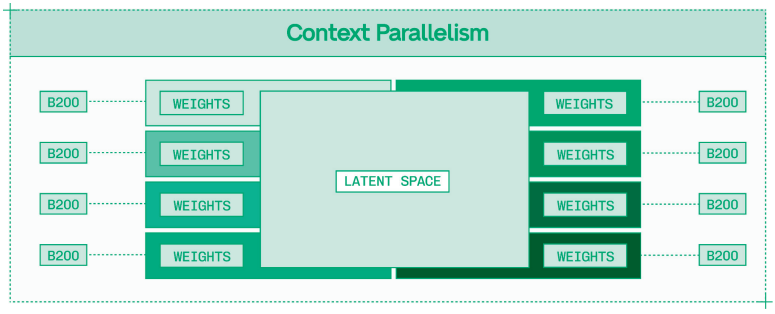


Figure 6.11: Context Parallelism replicates model weights but shares latent space to compute attention during video model inference.

Attention for transformer models is multi-head, usually with eight or more heads. Attention heads are independent, so they can be run separately with the results combined afterward.

Attention isn't the only thing that can be parallelized. For example, the latent decoding step using the variational autoencoder takes three to five percent of the total inference time and can be run across GPUs.

These parallelism techniques make AI video feasible. As video sequences get longer and video models get larger, parallelism will continue to be the most critical technique for video generation model inference.

CHAPTER 7

Production

Production

The purpose of inference engineering is to make generative AI models faster, less expensive, and more reliable to operate, which in turn lets you build better products.

This promise is only fulfilled if your inference engineering work makes it to production and scales alongside the hypergrowth and viral spikes that successful AI products generate.

When you scale production traffic, your assumptions are rigorously tested. Everything from sequence shapes to traffic patterns to what topic a user decides to chat about impacts your observed performance in production. And maintaining secure, robust infrastructure is an entirely different skillset from optimizing model inference on the GPU.

No matter how fast and efficiently a single instance can serve a model, with enough traffic, the service will be overwhelmed. That is not a PyTorch problem or a CUDA problem, it's an infrastructure problem, and requires a different mindset and different technologies.

Scaling in production introduces new complexities about where and how to get GPUs, balance traffic across them, and prevent downtime. Plus, cost accounting gets messy in the transition from paying per million tokens to paying directly for infrastructure.

Latency in production comes from more than just prefill and decode. You need to evaluate your system end-to-end and eliminate inefficiencies in the server, the network, and even the client in situations where you can own or influence client code.

This chapter introduces the essential considerations for scaling low-latency, high-throughput inference in production. And at the end, I'll invite you to try Baseten for deploying mission-critical inference workloads.

7.1 Containerization

Containerization is the practice of packaging an application together with its dependencies to standardize deployment in production. Containers

turn a program into a packaged artifact that can run anywhere – no more "it works on my machine."

Containers are lightweight because they share the underlying host operating system kernel (in this context, a kernel refers to a Linux kernel, not a CUDA kernel). This makes containers well-suited for packaging inference services.

For most developers, containerization is synonymous with Docker. Working with containers introduces more specific terminology:

- **Container:** An actively running environment that isolates an application and its dependencies.
- **Image:** An executable package that contains everything you need to run a piece of software.
- **Dockerfile:** A human-readable file with well-specified, machine-interpretable instructions for creating an image.
- **Registry:** A central repository for managing, storing, sharing, and distributing images.

NVIDIA, several cloud providers, and Docker themselves all operate container registries. A popular registry for AI is Docker Hub – Docker Hub is to images as Hugging Face is to model weights or PyPi is to Python packages.

Docker containers are composed of layers. You can take a pre-configured base image and add other layers with additional filesystem changes on top of it.

There are three types of layers:

- **Base image:** Either an operating system distribution like Ubuntu or a more complex image taken from a registry. The base image itself is composed of multiple layers.
- **Additional layers:** Filesystem changes including dependencies, application code, and configuration files as specified by Dockerfile instructions.
- **Container layer:** A thin, ephemeral layer created at runtime. Any changes to the running container, like creating, updating, or deleting files, are written to this layer and are lost when the container terminates.

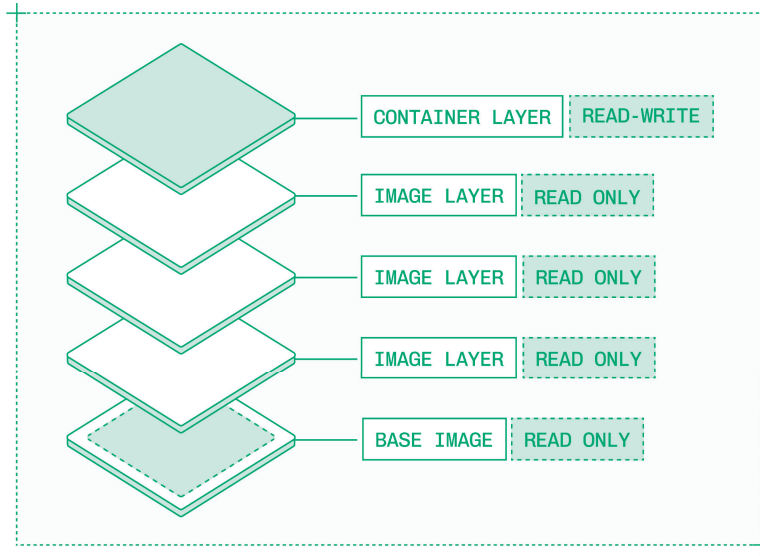


Figure 7.1: Docker containers are composed of layers, from a base image up to an ephemeral, writable top layer.

Inference engines like vLLM and SGLang offer official base images for active releases. It's generally a good idea to start from one of these proven images, rather than building your own from scratch.

7.1.1 Dependency Management

Dependency chains for inference are long and fragile. Getting to a working build is hard, which makes containerization essential for preserving a known good build in an ecosystem where breaking changes are all too common.

Images are built for a specific GPU architecture and model. A container includes many runtime components:

- **CUDA toolkit version:** The specific versions of CUDA, cuDNN, and drivers compatible with the rest of your stack.
- **Python packages:** Dependencies like `torch`, `transformers`, and `diffusers`.

- **Inference engine:** The version of vLLM, SGLang, TensorRT-LLM, or any other inference engine used.
- **System packages:** Linux packages like `ffmpeg`, especially common when working with audio, image, or video models.

Like a hiker on a backpacking trip, you want to pack light. Images built for inference are often many gigabytes. For fast deployment and efficient operation, only include strictly necessary dependencies.

Another best practice is pinning versions. Having a pinned dependency tree keeps the system runtime behavior consistent across different environments and enables repeated builds of the image with the same result. Specify exactly which version of each dependency should be included in the image.

```
requirements:
  transformers          # No
  transformers>=4.40.0  # No
  transformers==4.56.2  # Yes
```

Figure 7.2: Requirements should be pinned to exact versions to prevent future changes from breaking inference containers.

Tools like uv, poetry, or pip will flag any version incompatibilities and throw an error when building an image. With pinned versions, once an image is successfully built once, it will always resolve dependencies to the same versions and protect you against breaking changes.

Breaking changes are particularly common when working with newly released models. When new versions of models like DeepSeek are announced, the entire inference ecosystem races to offer day zero support.

When building images for brand-new models, inference engineers often rely on overnight builds or other developer pre-releases for dependencies rather than stable releases. These early versions are more prone to bugs and often need to be rebuilt on stable releases in the days and weeks following the model drop.

7.1.2 NIMs

NVIDIA Inference Microservices (NIMs) are pre-built Docker containers for popular open models.

Containers make inference service implementations portable. NVIDIA created two types of NIMs:

- **Multi-LLM NIM:** A flexible container for running a family of models on a supported GPU architecture.
- **LLM-specific NIM:** An engine optimized for a specific model on a specific GPU configured for maximum performance.

NIMs are available for various common combinations of model, GPU architecture, GPU count, and configuration.

A NIM is like any other container. You can use a NIM as a starting point to build on, as a reference architecture to learn from, or as an out-of-the-box inference service.

However, if you're looking for maximum control instead of a done-for-you configuration, you'll generally be better off building your own container from a less opinionated base image rather than adapting a NIM.

7.2 Autoscaling

The goal of autoscaling is to ensure that you always have enough resources to serve all incoming requests while maintaining your latency SLAs without wasting money on idle GPUs.

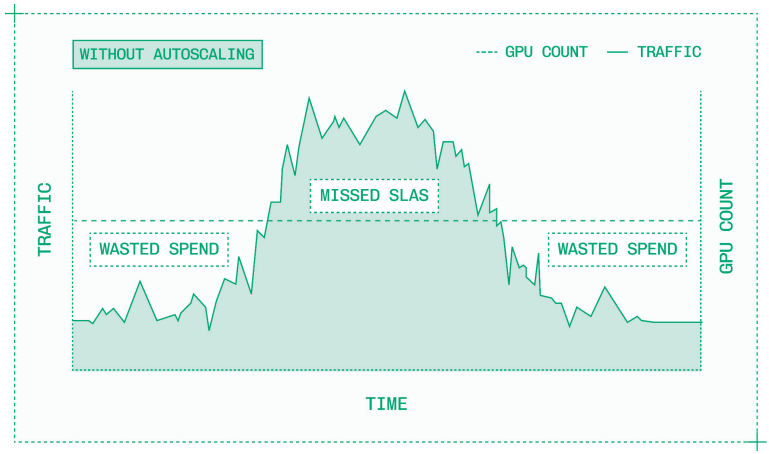


Figure 7.3: Without autoscaling, inference systems waste resources during traffic lulls and miss SLAs during traffic spikes.

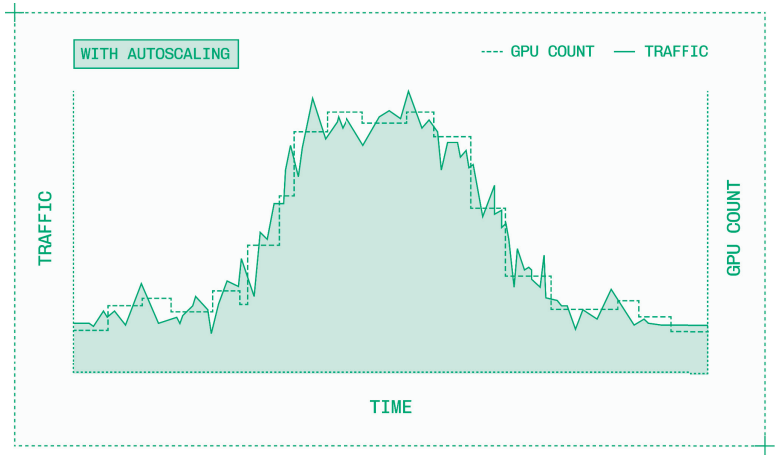


Figure 7.4: A strong autoscaling system for inference matches resources to demand.

Autoscaling systems use Kubernetes, an open-source container orchestration system, along with a cluster-level system for provisioning and deallocating compute. Kubernetes can run one or more replicas of a model container, each on its own instance. An instance includes the GPUs and other hardware resources that the container requires.

Kubernetes works by composing a group of hardware resources together into a cluster. This cluster has two types of components:

- **Control plane:** Makes routing and scaling decisions.
- **Worker plane:** Runs the actual containerized applications.

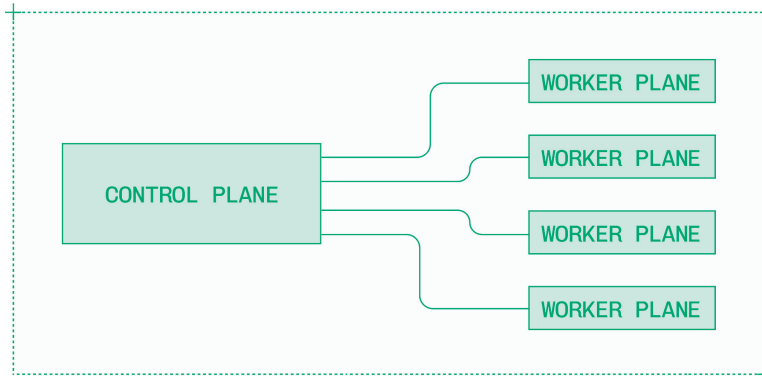


Figure 7.5: Kubernetes clusters have a single control plane that orchestrates multiple workers.

A Kubernetes cluster can run multiple replicas of multiple models. But how do you decide how many replicas of each model to run? Unless your traffic is unusually consistent, there probably isn't one number of replicas that perfectly matches your needs.

Autoscaling is the practice of dynamically adjusting the number of replicas allocated to a given model within a cluster. There are two ways to make autoscaling decisions:

- **Utilization:** Scale up and down based on GPU utilization signals like memory usage or compute usage.
- **Traffic:** Scale up and down based on the number of requests being processed in the system.

Utilization and traffic don't always match. For example, in LLM prefill, a few requests with hundreds of thousands of uncached input tokens could cause much higher utilization than many small requests with high cache hit rates.

Traffic-based scaling decisions can be made proactively, while utilization is a lagging indicator. Use both in combination to keep system resources matched with demand.

When designing a traffic-based autoscaling system, you want to configure five factors:

- **Min replicas:** What is the minimum number of replicas that stay running, regardless of traffic?
- **Max replicas:** What is the maximum number of replicas that you can allocate when traffic is high?
- **Autoscaling window:** How long is the sliding timeframe that you use to measure traffic and make autoscaling decisions?
- **Scale down delay:** How long after a scale down is suggested do you wait in case of another traffic spike?
- **Concurrency target:** How many requests can each replica handle at once?

The exact configuration determines how well the autoscaling system achieves its goals of maintaining latency SLAs without wasting resources. For example, increasing the scale down delay prevents premature scale-downs for spikey traffic, but could result in unnecessary spend after traffic truly cools off.

7.2.1 Concurrency and Batch Sizing

To properly operate a traffic-based autoscaling system, you need a strong understanding of how much concurrent traffic each instance can handle.

Most model inference services can handle more than one request at a time via batching. There are several approaches to batching:

- **Static batching:** The server waits until the batch is full before starting inference.
- **Dynamic batching:** The server waits until the batch is full or a configured amount of time has passed before starting inference.
- **Continuous batching:** The server continuously runs inference, swapping in requests as slots become available.

Inference engines like vLLM, SGLang, and TensorRT-LLM implement robust continuous batching (or in-flight batching as TensorRT-LLM calls it), where requests are batched at the token level. This minimizes latency relative to static batching.

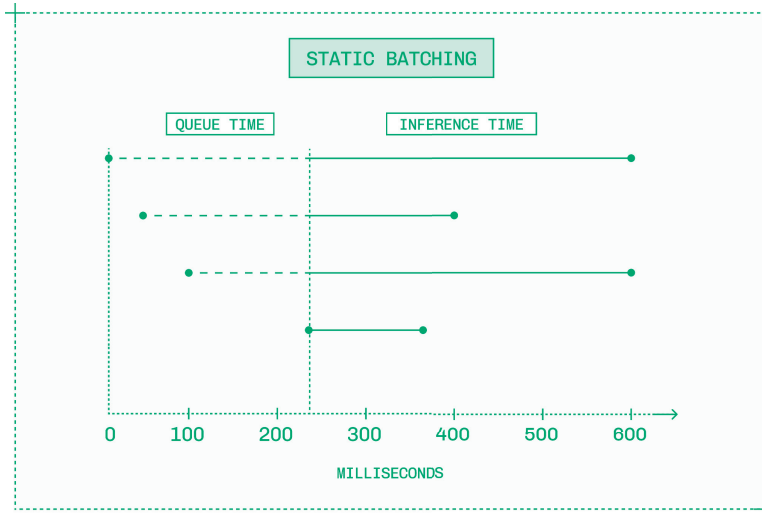


Figure 7.6: Static batching sets a fixed batch size and waits for the batch to fill before beginning inference, leading to long wait times for early requests.

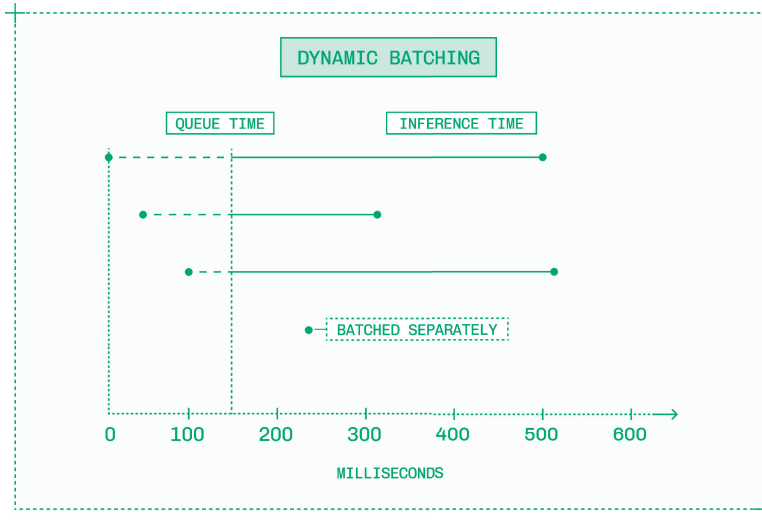


Figure 7.7: Dynamic batching adds a cutoff time after which a batch is run whether or not it is full.

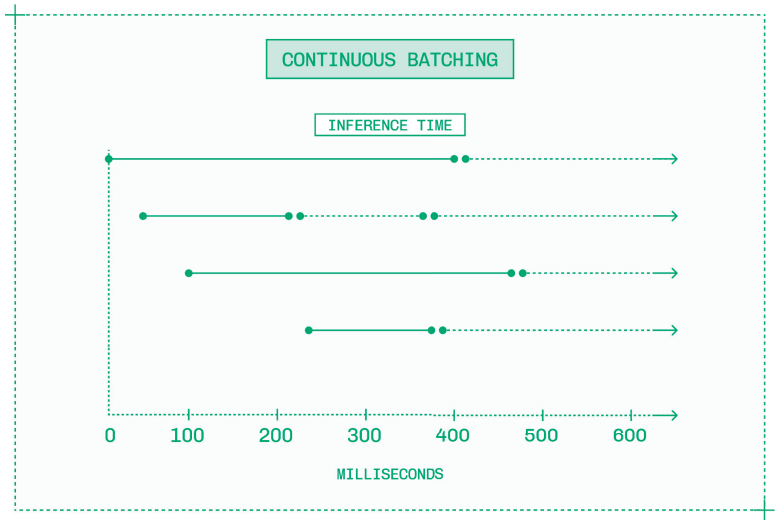


Figure 7.8: Continuous batching operates at the token level, switching in new requests as old requests finish.

Batch sizing trades off latency for throughput. Increasing the batch size will produce more throughput overall, but each user's latency will get worse. Test performance across multiple batch sizes to find the right fit for your model, instance, latency target, and budget.

This is controlled at the autoscaling configuration level via the concurrency target and at the replica level via the batch size, which should match.

Once every active replica reaches its maximum concurrency, the autoscaling system knows to spin up more replicas. If enough replicas are kicking off half-full batches, it's time to scale back down.

7.2.2 Cold Starts

A cold start is the time it takes to spin up a new replica of a model.

The overall performance of an autoscaling system depends on its cold start speeds. If you can't spin up replicas fast, it's hard to confidently scale down, leading to over-provisioning.

There are several factors that affect cold start times:

- **GPU procurement:** How quickly can you add the necessary GPUs to your cluster and allocate them to the model?
- **Image loading:** How quickly can you load the container image onto the newly procured instance?
- **Model loading:** How quickly can you load the model weights into the container?
- **Engine startup:** How quickly can you start your inference engine, including any compilation time?

Each of these factors needs to be optimized separately.

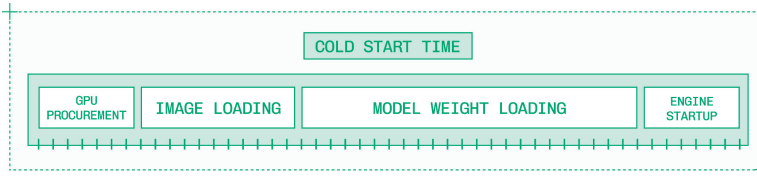


Figure 7.9: Each step in the cold start process adds to the overall timeline.

Unless you have a pool of warm nodes that you're flexing between models, GPU procurement speed is mostly a function of your cloud provider. Section 7.3.1 covers procuring GPUs, and node start time is one of the negotiable factors in a contract.

However, engineers can do a lot on loading images and weights and starting containers and engines.

Loading images and model weights is a function of how quickly you can write gigabytes, often hundreds of gigabytes, of data onto your instance. There are two ways to load images and weights faster: make them smaller or get more bandwidth.

Including only necessary steps and dependencies makes images smaller and faster to build into containers, while quantizing model weights has the additional benefit of making them faster to load during cold starts.

For small models, the strategy used to be baking the weights into the image to simplify caching and loading. However, now that most models have dozens or hundreds of billions of parameters, the weights dwarf the image and are better loaded separately.

Where you load weights from has a massive impact on the bandwidth. If you're loading from a third party like Hugging Face, you're limited by their egress speed. And storing your weights in an S3 bucket introduces network latency and data transfer costs.

For loading multi-hundred-billion-parameter models, you need gigabytes per second of bandwidth. The best way to get this is by loading over network within a node from a source cached physically near the GPU instance within the same datacenter.

Inference engines like vLLM and SGLang are fast to start up. But engines like TensorRT-LLM and optimized models with PyTorch have a compilation step that targets the specific hardware resources to build the model inference engine. These compilations often take several minutes.

In these cases, caching built engines massively improves cold start times. Both TensorRT-LLM and PyTorch have image caching mechanisms that make this possible, though you'll always need to load a cached engine into an instance with exactly the GPU type, CUDA version, and software dependencies as the environment that the engine was built in for it to run properly.

7.2.3 Routing, Load Balancing, and Queueing

Once there are multiple replicas online, the system needs to make a decision about which requests to send to which replicas. There are two types of components that make these decisions:

- **Routers:** A router works at the request level to determine the ideal place to send a given request. A router answers "where should this request go?"
- **Load balancers:** A load balancer works at the system level to even out requests between multiple options. A load balancer answers "where could this request go?"

In complex systems, there isn't just one router and one load balancer. Routing occurs throughout the stack with load balancers injected at key points to keep system-wide performance stable.

Overall, you want to split load equally across replicas. However, routing and load balancing are not as simple as saying, "Well, I have 3 replicas and 12 requests, so let's put 4 requests on each replica."

Each request may have a different number of input tokens. If most requests have 100 input tokens, a request with 10,000 will unbalance a simple system. Some requests are also better handled by certain replicas. Examples include:

- **KV cache-aware routing:** Direct a request to a replica that already has a matching prefix in its KV cache.
- **LoRA-aware routing:** Direct a request to a replica that already has the desired LoRA fine-tuned weights in memory.

Intelligent request routing uses information from the inference engine and any orchestrators like NVIDIA Dynamo to route requests based on sequence length, prefix, and LoRA needs.

Load balancing and routing are not enough. When an autoscaling system receives more traffic than it can handle, it needs a way to hold onto requests as it scales up more resources or waits for existing resources to become available.

A queue is the infrastructure primitive for handling this situation. A standard queue is a first-in, first-out system for excess requests, though you can do a more complex implementation like a priority queue to, for example, give paid users priority over free users in high-traffic scenarios.

As new replicas come online, ensure that the queue sees them and requests don't continue waiting for the existing replicas. Each new replica should immediately be assigned up to its concurrency limit in queued traffic once active.

7.2.4 Scale to Zero

Advanced autoscaling systems implement a mechanism for scale to zero, where the system can scale down to zero active replicas if there is no traffic, then scale up automatically when traffic is received.

Scale to zero relies on two prerequisites:

- **Fast cold starts:** As users may be waiting live, cold starts must be as fast as possible.
- **Robust queueing:** The system needs to be able to hold the incoming requests until a replica is live.

Even with these capabilities, scale to zero is not a fit for all workloads.

Scale to zero is great for development, when testing is bursty and latency for the first request is unimportant. And in production, scale to zero is useful for applications that only get traffic periodically, like an agent that's only accessed during business hours in one country or an offline system designed for daily batch processing jobs.

However, if you're relying on scale to zero to keep costs low in a latency-sensitive application that gets light, unscheduled traffic, it's probably a sign that your AI application is not yet ready for dedicated infrastructure and should use pay-per-token APIs until greater scale is reached.

7.2.5 Independent Component Scaling

AI applications are increasingly built on multi-model, multi-stage compound AI workloads where inference engineers need to coordinate multiple steps to fulfill a single request.

These steps may have different hardware needs. A voice activity detector model needs a far less powerful GPU than the transcription model it's chunking data for, while the LLM processing that transcript may need a full node with multiple GPUs. And the scaling parameters for each step in the pipeline also differ.

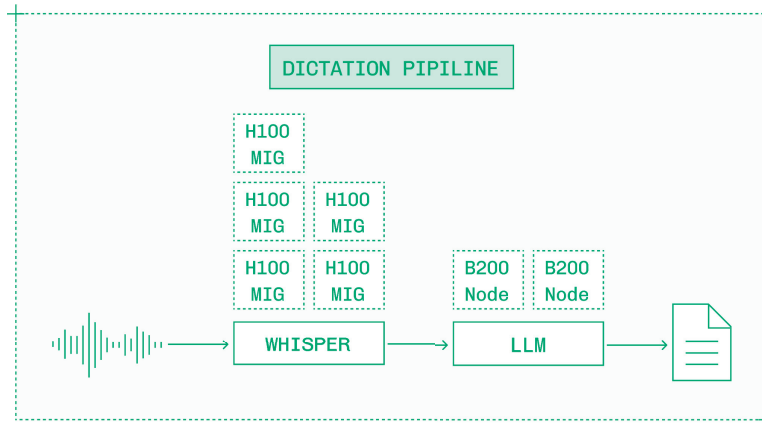


Figure 7.10: Independent component scaling gives each model access to appropriate resources and individual scaling.

For these pipelines, you need to decompose autoscaling decisions and scale each step individually to right-size resources per step and avoid both bottlenecks and overprovisioning.

However, every model in the pipeline should run in the same cluster. If it takes 10 milliseconds to send a message within a cluster and 50 milliseconds to send messages between clusters, that 40-millisecond difference across a 5-step pipeline would be 20 percent of a one-second latency SLA.

7.3 Multi-Cloud Capacity Management

Autoscaling within a single cluster works up to a certain point. But high-volume deployments serving a global user base need thousands of GPUs distributed around the world.

It's straightforward to build multi-cloud inference as a collection of siloed compute across different cloud providers. But in these setups, there's no way to use inter-cloud compute fluidly, and moving workloads across clouds is a tedious, error-prone process.

True multi-cloud inference requires building a multi-region, multi-provider bin packing tool, which treats distinct pools of compute as fungible with

each other. Like Kubernetes within a single cluster, multi-cloud capacity management must take a global view, enabling self-healing and global scheduling.

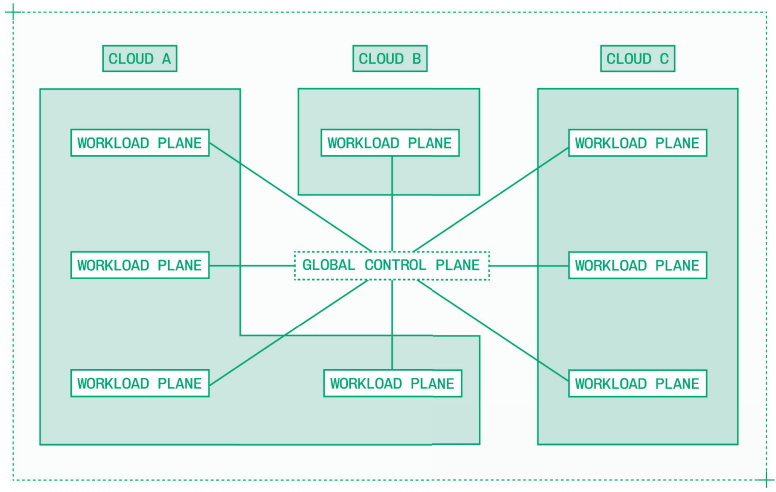


Figure 7.11: A multi-cloud approach extends the idea of control and workload planes to a multi-cluster, multi-region system.

Running true multi-cloud inference unlocks:

- **Capacity:** Pool capacity from multiple providers for greater and more flexible GPU access.
- **Redundancy:** Split inference across providers for resiliency against outages.
- **Latency:** Run inference close to your end users to reduce network latency overhead.
- **Compliance:** Run inference in compliance with data sovereignty and other regulatory requirements.

Scaling from one cluster in one cloud to many clusters in many clouds requires a new coordination layer. A multi-cloud architecture contains:

- **Control plane:** Handles model deployment and global scaling decisions, receives real-time event streams.

- **Workload planes:** Handles direct inference traffic and in-cluster scaling decisions, reports utilization and demand.

This separation of responsibilities ensures that individual workload planes can serve traffic independently. If something happens to the control plane or any given workload plane, other workloads should be unaffected.

7.3.1 GPU Procurement

There are a good number of companies in the business of providing access to GPUs. The three major types are:

- **Hyperscalers:** Large cloud providers like AWS or GCP.
- **Neoclouds:** GPU-focused clouds like Coreweave or Nebius.
- **Resellers:** Secondary markets like SF Compute Company.

Players in this space vary in their capacity, availability, and reliability. You generally pay a premium for hyperscalers and across all providers there is a tradeoff between cost and factors like uptime SLAs, support, regional availability, instance configuration, and cluster sizes.

The first challenge is securing capacity. It is often difficult to get your hands on the GPUs you need, especially the latest hardware. Large clusters are also hard to get, with relatively few players offering blocks of hundreds of nodes.

Many cloud providers allocate the majority of in-demand GPUs to their largest customers on long-term reservations. You may need to work across multiple cloud providers to get the GPUs you need in the right regions.

Cloud GPUs can be procured via three different mechanisms:

- **Reserved:** Blocks of hundreds or thousands of GPUs are reserved for months or years at discounted rates.
- **On-demand:** Individual instances are available as needed up to a given quota for a relatively high per-hour cost.
- **Spot:** Discounted on-demand instances that can be pre-empted at an agreed-upon notice period, often minutes.

Large-scale inference generally uses a blend of GPU sources, with a baseline of low-cost reserved instances and a mix of on-demand and spot for handling peaks in traffic. These GPUs are distributed across multiple clusters worldwide for proximity to end users.

7.3.2 Geo-Aware Load Balancing

Successful AI applications have users all over the world. Just like an individual cluster has a load balancer to ensure that every GPU in the cluster receives the right amount of traffic, a multi-cluster system needs a global load balancer.

You don't want a user request sitting around in some queue when there is spare capacity elsewhere, but you also don't want to make a habit of sending a request from Singapore to a server in San Francisco.

As a rule of thumb, it takes five milliseconds for a request to pass through a time zone. So, sending data from New York to San Francisco takes fifteen milliseconds one way. Given how small latency budgets are, it's important to run workloads as close to end users as possible.

7.3.3 Building for Reliability

GPUs are infamous for their high failure rate in production. Every engineer who has done a large-scale training run knows that they need to account for the eventuality that hardware will fail.

For example, in their Llama 3 paper, Grattafiori and colleagues revealed that while running 16,000 GPUs for a period of 54 days, the Llama team experienced 419 unexpected interruptions, primarily due to hardware failure. This works out to approximately one failure per 50,000 GPU-hours.

50,000 hours might sound like a long time, but running a single node of eight GPUs for inference for an entire year is over 70,000 GPU-hours. Inference engineers should expect hardware failure.

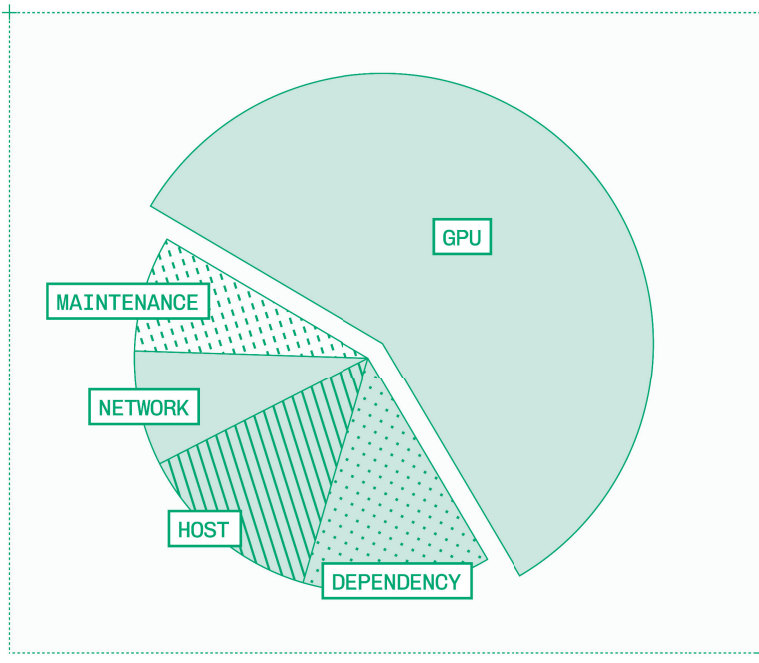


Figure 7.12: Root cause of failures when training Llama 3, adapted from "The Llama 3 Herd of Models" (Grattafiori et al., 2024).

GPU health is a node-level concern. When a single GPU fails, other GPUs on the node often fail next or need to be taken offline for maintenance. Proactively noting failures, cordoning nodes, and cycling pods keeps individual clusters healthy.

GPU failures aren't the only thing that can bring down inference. Cloud providers have scheduled maintenance and their own unscheduled downtime. Every layer of infrastructure must be reinforced to provide high reliability.

Multi-cloud inference brings two new approaches to high reliability:

- **Active-active:** A high-availability posture where multiple regions or clusters actively serve live traffic at the same time. If any plane fails, traffic seamlessly continues on the others.
- **Active-passive:** A failover posture where a "hot standby" cluster or region is kept ready but idle. If the active plane fails, traffic is cut over to the passive plane.

When individual clusters, regions, or cloud providers go down, seamlessly failing over to another workload plane keeps reliability high and latency low.

7.3.4 Security and Compliance

Cloud infrastructure has been a hot topic for security and compliance departments for more than twenty years. For AI models to power mission-critical applications, inference must be both secure and compliant.

Security and compliance conversations generally center around three areas:

- **User data:** Security and compliance departments want to ensure all data, including user inputs and model outputs, is protected.
- **Model weights:** For companies with fine-tuned or proprietary models, the weights are an invaluable trade secret.
- **Infrastructure:** GPUs themselves and access to intelligence are both targets for abuse.

One of the easiest decisions you can make to improve security is to simply not store user inputs or model outputs. This may not be possible – you might have logging requirements or user agreements to retain usage data for future model training – but if you don't need to retain user data, you can reduce your attack surface.

Securing AI inference workloads and associated data is similar to securing any other containerized workload. Data encryption, container security, network and access controls, and workload isolation, all validated by extensive third-party penetration testing, remain the gold standard.

Increasingly, inference engineers need to support applications running in regulated industries and compliance-heavy regions. One place where multi-cloud infrastructure helps is that in order for your application to comply with a certification like SOC 2 Type II or a regulation like HIPAA, your providers generally must also be compliant. In this case, being able to move workloads to compliant providers is useful.

Another benefit of multi-cluster infrastructure is running one model across multiple regions. Certain industries and countries have data residency

requirements, where user data from their country cannot be processed on servers in a different country.

For example, having one cluster in a provider near Toronto and another in a provider near New York lets you keep Canadian data in Canada and American data in the United States while providing minimal latency overhead to users across the geographic region.

7.4 Testing and Deployment

In addition to any replica-level testing and benchmarking performed while configuring the inference engine, it's important to test systems end-to-end before deployment.

There are several strategies for testing inference:

- **Manual testing:** Writing scripts (or clicking buttons) to send synthetic traffic to an inference service.
- **Load testing:** Automatically sending a large volume of traffic to test a system's ability to scale and maintain performance.
- **Shadow traffic:** Copying live traffic to test deployments to measure performance under real-world conditions.

Testing inference services is expensive. It takes engineering time to configure the tests and measure the results, and it takes GPUs to run inference for the test traffic. To a degree, that's just the cost of doing business, but think carefully about how to minimize testing expenses. For example, shadow traffic testing could start with copying a random sample of production traffic, followed by a shorter-duration load test.

When testing, keep in mind that AI product usage generally fluctuates on daily and weekly cycles.

Once you're confident in the performance and stability of your updated inference system, it's time to deploy to production.

7.4.1 Zero-Downtime Deployment

Inference engineers use high-availability deployment strategies to avoid downtime.

A traditional high-availability design is a blue-green deployment. In this setup, there are two identical environments: the original blue deployment and a new green deployment running the updated service. Once the green environment is ready, the full traffic load cuts over from the blue to the green environment, with the blue environment staying ready for rollback in case of issues.

However, blue-green is not well suited for large scale inference workloads due to the same GPU capacity and cost issues that make large-scale testing difficult. If the blue deployment is using 100 GPUs, the green deployment requires another 100 GPUs before traffic can cut over.

Instead, inference engineers can get similar benefits with lower GPU overhead using canary deployments. Inspired by the canaries that were used to detect gas in coal mines, a canary deployment catches errors before they affect large numbers of users.

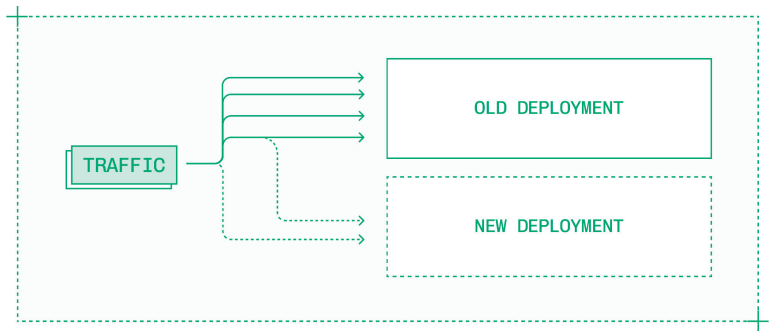


Figure 7.13: Iteratively shifting traffic over to the new deployment prevents multiple issues during inference service updates.

A canary deployment is a 4-step process:

1. Build a new deployment of the inference service and get it ready to handle incoming requests.

2. Direct a small percentage of the incoming live traffic to the new service.
3. Monitor the new service and ensure it is handling traffic correctly. Revert if there are any issues.
4. Gradually increase traffic, while monitoring for issues, until the new deployment handles 100 percent of traffic.

These canary deployments can be rolled out quickly, with just a few minutes of traffic ramp, or ramped slowly to ensure stability at each stage. And with autoscaling, canary deployments don't increase cost much at scale because reducing traffic to the production system causes it to scale down some replicas.

With autoscaling, the new deployment will default to the minimum number of replicas when there is no traffic. Throughout the canary deployment process, ensure that the new deployment has enough active replicas to properly handle requests. Otherwise, users will see a latency spike as their requests are queued until autoscaling completes.

7.4.2 Cost Estimation

Switching from consuming tokens from a public API to doing your own inference on dedicated GPUs requires changing how you think about cost.

Cost on public APIs is simple: a price per million tokens times the number of tokens you use. There are a couple of variables – cache hits versus cache misses for input tokens, discounts for high-volume users – but cost remains a linear function of usage.

One motivation for investing the time and effort in inference engineering is to take control of your unit economics and escape per-token pricing. But it's a difficult mental transition.

The blessing and curse of dedicated inference is that cost is now a function of many variables. This is good because it gives you control, but it makes estimation difficult. Factors that affect cost include:

- **Batch sizing:** Is the deployment optimized for latency with low batch sizes or throughput with high batch sizes?

- **Traffic patterns:** Is traffic consistently saturating active GPUs, or is capacity going spare?
- **Sequence lengths:** How many input and output tokens do requests have both on average and in outlier cases?

Given this complexity and the difference in cost between input and output tokens, it's generally more productive to convert your token price into a total cost and compare that to dedicated instead of trying to reverse engineer a per-token price from what you pay for GPUs.

```
# Equation for estimating total cost from per-token APIs
total_input_tokens = 1000  # In millions
total_output_tokens = 500  # In millions

price_per_million_in = 1.25
price_per_million_out = 10

input_cost = (total_input_tokens * price_per_million_in)
output_cost = (total_output_tokens * price_per_million_out)

total_cost = input_cost + output_cost

# Total cost: $6250.00
```

Figure 7.14: An equation for estimating the total cost of using per-token APIs in a product.

```
# Equation for estimating total cost from dedicated deployments
total_gpu_hours = 1600
price_per_gpu_hour = 3.50

total_cost = total_gpu_hours * price_per_gpu_hour

# Total cost: $5600.00
```

Figure 7.15: An equation for estimating the total cost of using dedicated deployments in a product.

Cost estimates should use a long time horizon, ideally at least a week, to smooth out variations in usage.

The other factor to consider in dedicated deployments is the cost of engineering time spent building and maintaining inference systems. This investment, while justified in increased reliability, security, and control, should be added to the GPU costs to form a complete picture around total cost of ownership (TCO) for inference.

7.4.3 Observability

Inference is mission-critical, so it must be monitored like any other mission-critical component of an application, with alerting, logs, and observability built at the right level of abstraction.

The first question is what to monitor. Inference observability includes measuring:

- **Total volume:** The number of requests that a model deployment is receiving.
- **Request and response sizes:** The input and output sequence lengths for the requests being processed.
- **Response codes:** The count of 2XX, 4XX, and 5XX response codes issued by the model server.
- **Latency:** Metrics like time to first token, tokens per second, and end-to-end latency on a P50, P90, and P99 basis.
- **Replica count:** The number of instances actively serving traffic, and the number of instances starting up, if any.
- **Utilization:** The amount of utilization across CPU, host memory, GPU, and GPU memory.
- **Queue depth:** For systems with asynchronous traffic, the number of requests enqueued and waiting to be processed.

These metrics are interdependent. A spike in latency could come from request volume, but it could also come from long input sequences. Seeing these metrics together lets inference engineers understand not only what is happening but also why.

When things go wrong, inference engineers need information to fix issues. Logs, both server logs and audit logs showing changes to an inference service, deliver that information in real time.

Observability cannot be siloed. When you build observability for inference, build it with deep integration into existing observability and alerting tooling – Grafana, Datadog, PagerDuty, Sentry – to put inference information in context with the rest of the application.

7.5 Client Code

Inference engineering draws on many technologies, from CUDA to Kubernetes. But there's one critical area that's often overlooked when optimizing for latency and building for scale: client code.

There are two sides of a call to an inference service:

- **Client:** The browser, agent, or application making a request to the inference engine.
- **Server:** The inference service that handles the client request and returns the model results.

The industry standard for client code is the OpenAI SDK, which supports a wide range of compatible providers in addition to OpenAI's own models. Popular AI engineering frameworks and libraries like LangChain, Vercel AI SDK, LiteLLM, LlamaIndex, and dozens more can also serve as clients.

Whether you're using an existing library or your own code, there is the potential for latency overhead or throughput bottlenecks. And for real-time applications, you may need a protocol other than HTTP, like WebSockets, to deliver a continuous connection.

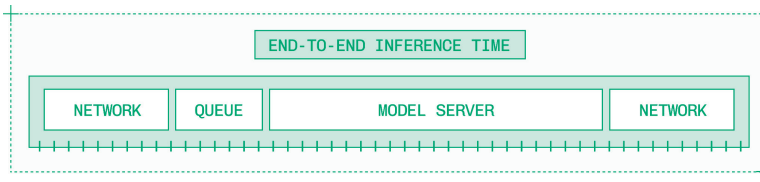


Figure 7.16: On-server inference time is just a fraction of the end-to-end latency for a given request.

7.5.1 Client Latency Overhead

Depending on the client's internet connection and the protocol used, establishing a session between a client and server takes a few dozen milliseconds.

In a high-performance system with a 300-millisecond P95 end-to-end latency SLA, a TLS handshake costs at least ten percent of that latency budget before inference even starts. Future requests from the same client should save time by re-using existing sessions.

Session re-use is not a new idea by any means, and tools like the OpenAI SDK provide it silently under the hood. However, when building your own client for non-standard modalities, follow best practices like session re-use.

7.5.2 Asynchronous inference

Some systems are built for throughput, not latency. Use cases like bulk document processing and corpus embedding are not latency sensitive, so it makes sense to switch to asynchronous jobs.

Asynchronous requests are a "fire and forget" approach to executing inference.

Ordinary synchronous inference requests have a timeout, generally of a few minutes, after which the request will fail. Asynchronous jobs fix this by immediately acknowledging the request and later returning the result of the asynchronous job to a webhook supplied in the original request.

Asynchronous jobs still have time limits, but these requests are usually measured in hours, not minutes. Along with strong server-side queuing, asynchronous requests make high-throughput, latency-insensitive systems more robust and efficient.

7.5.3 Streaming and Protocol Support

Streaming makes applications feel instant. For language models, streaming text output over HTTP is sufficient. But for other modalities, especially

live voice and video, both input and output streams need to be able to carry more data.

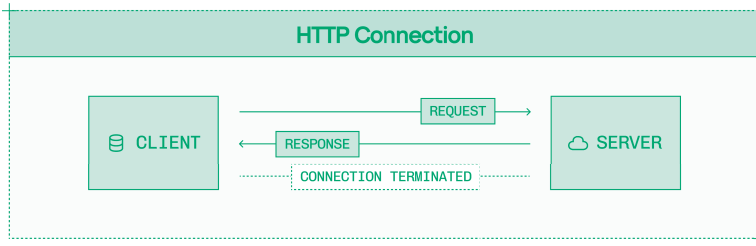


Figure 7.17: One-time HTTP requests and responses are a good fit for use cases like text chat, but not for continuous streaming.

The two most common bi-directional streaming client-server connection protocols are:

- **Websockets:** For streaming use cases where strong schema enforcement is not required.
- **gRPC:** For well-defined service-to-service communication.

WebSockets are useful for transmitting unstructured and real-time data, like audio, where the server receiving the request can parse it and process it downstream. With WebSockets, a server can support up to a fixed, developer-configurable number of clients; when that concurrency is reached, new connections cannot be established and must wait until either a slot is free or another replica scales up.

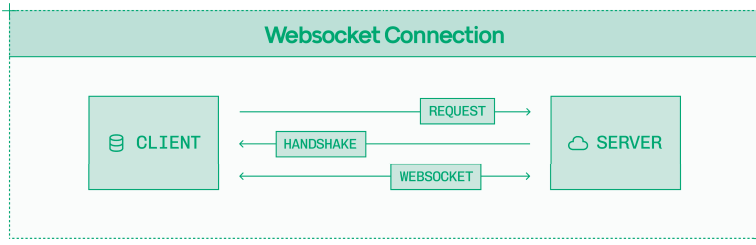


Figure 7.18: WebSockets establish a continuous connection for unstructured data like audio streams.

Similar to WebSockets, gRPC enables bi-directional streaming support, but for structured data. Requests transmitted via gRPC must follow a predefined schema, which takes away the load of having to parse the input. This additional validation layer makes gRPC slightly slower than WebSockets.

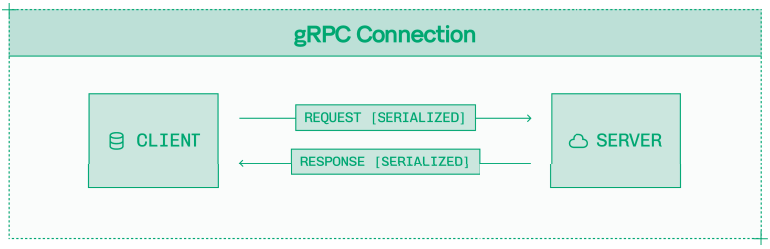


Figure 7.19: gRPC establishes a continuous connection for well-defined service-to-service communication.

7.6 Production Inference with Baseten

This book contains everything I've learned about inference in four years of working at Baseten.

Baseten is an AI infrastructure company founded in 2019. At Baseten, we are focused on highly performant and highly available inference for both open models and custom models. We also offer a platform for pre-training, post-training, and reinforcement learning.

We run inference for the world's fastest-growing startups and most innovative enterprises, including Cursor, World Labs, Notion, OpenEvidence, Clay, Abridge, Gamma, Ambience, Writer, and hundreds more.

At Baseten, we focus on four essential pillars to deliver the fastest mission-critical inference:

- **Performance:** Consistent low latencies at scale powered by the Baseten Inference Stack.
- **Infrastructure:** Reliable multi-cloud deployments, fast and granular autoscaling, and robust security.
- **Tooling:** An intuitive and productive developer experience with logging, observability, and programmatic access.
- **Applied expertise:** Hands-on-keyboard implementation and assistance from forward deployed engineers.

We would be honored to provide fast, reliable inference for your AI-powered products.

Also, we are continuously hiring for all roles across engineering, sales, marketing, and operations. To learn more, visit https://baseten.com/careers.

APPENDIX A

Inference Glossary

Activation function: A (mostly) differentiable nonlinear function like ReLU inserted between linear layers to prevent multi-layer neural networks from collapsing into a single matmul.

Active–active: A high-availability posture where multiple regions/clusters actively serve live traffic at once. If any plane fails, traffic seamlessly continues on the others.

Active–passive: A failover posture where a "hot standby" cluster or region is kept ready but idle. If the active plane fails, traffic is cut over to the passive plane.

Ada Lovelace (architecture): NVIDIA's graphics-oriented GPU architecture, released alongside Hopper in 2022. Useful for small models and cost-sensitive workloads, not suited for large-scale LLM inference.

Agent: An AI application that takes action rather than just providing information. Agent workflows usually rely on multiple inference calls, often across multiple models and modalities, and require access to tools.

AI-native application: A product where the core UX and value depend on generative models. Inference choices are downstream of app constraints: modality, latency budget, unit economics, and usage patterns.

Ampere (architecture): An older NVIDIA GPU architecture still used in legacy or small-scale deployments. Hopper and Blackwell architectures generally outperform Ampere on both raw speed and cost at scale.

Application Programming Interface (API): A structured interface for sending requests and receiving responses. Inference engines expose an API for making queries to models.

Arithmetic intensity: Operations performed per byte moved for a given algorithm. When compared to a GPU's ops:byte ratio, arithmetic intensity indicates whether a kernel is compute bound or memory bound.

Attention: The core transformer mechanism relating a token to prior tokens via Q/K/V projections and softmax. Attention is a primary target for optimization due to its compute and memory demands.

Automatic Speech Recognition (ASR): Audio-in, text-out transcription models (e.g., Whisper). Decoder work dominates runtime and benefits from LLM-style optimizations and in-flight batching.

Autoregressive token generation: Iterative generation of tokens where each token depends on each previous token. Autoregressive token generation is split into two phases: a prefill phase where input tokens are processed, and a decode phase where output tokens are generated.

Autoscaling: Scaling the number of replicas serving a given model up and down automatically based on traffic or utilization. Autoscaling matches capacity to demand, maintaining latency SLAs and minimizing wasted spend.

Autoscaling window: The rolling time horizon used to decide scale-up/scale-down actions. Longer windows keep replica counts steady; shorter windows react faster to spikes.

B200: NVIDIA Blackwell-based datacenter GPU with 192 GB of VRAM, 8 TB/s of memory bandwidth, and 5 petaFLOPS of FP8 compute.

B300: NVIDIA Blackwell-based datacenter GPU with 288 GB of VRAM, 8 TB/s of memory bandwidth, and 5 petaFLOPS of FP8 compute.

Bandwidth: The amount of data per second that can pass through memory like VRAM or an interconnect like NVLink.

Baselines: Initial, carefully recorded measurements of performance and quality before applying optimizations. Baselines enable clear attribution of gains or regressions.

Basic Linear Algebra Subprograms (BLAS): A standard interface for fundamental operations in linear algebra.

Batch: Process multiple inputs simultaneously, common for LLM inference.

Batch sizing: A core latency-throughput lever for inference engines. Larger batches improve total throughput but worsen per-user latency.

Benchmark (intelligence): A measurement of a model's ability to answer questions correctly or take appropriate actions (e.g., MMLU).

Benchmark (performance): A measurement of an inference service's latency and throughput for a given model with a defined workload.

BF16: A 16-bit floating-point format with larger exponent than FP16, useful in training and sometimes inference. Higher dynamic range helps preserve outliers.

Bin packing (multi-cloud): The practice of treating heterogeneous pools of GPUs across clouds, regions, and clusters as a single schedulable resource, enabled by multi-cloud capacity management infrastructure.

Blackwell (architecture): NVIDIA's late-2024 GPU generation featuring FP4 support, microscaling formats (MXFP8, MXFP4, NVFP4), and high memory bandwidth.

Blue-green deployment: Two parallel production environments ("blue" and "green"); shift traffic between them for zero-downtime deploys and quick rollback.

Cache-aware routing: Steering requests to replicas that already hold matching prefixes or required LoRAs. Higher cache hit rates yield lower TTFT.

Canary deployment: A new production deployment that starts with a small share of live traffic to a new deployment to validate stability and performance. Over time, the new deployment absorbs all production traffic.

Causal language model (CLM): A decoder-only transformer that predicts the next token given the prior context. All generative LLMs in this book are CLMs.

Central Processing Unit (CPU): A general-purpose processor optimized for sequential workloads. CPUs are used for orchestration, scheduling, networking, and preprocessing, but rarely handle generative AI inference directly.

Chat template: The model-specific formatting and serialization of messages (roles, separators, beginning/end of sequence tokens).

Chunked prefill: Splits long inputs into chunks and overlaps prefill with decode or other work, preventing single long sequences from monopolizing resources.

Classifier-free guidance: Balances unconditional and prompt-conditioned denoising passes on each step of image generation. Lower guidance enhances creativity; higher guidance enforces prompt adherence.

CLIP (text encoder): A text/image encoder used in earlier image pipelines (e.g., SDXL). Modern systems often swap in full LLMs for stronger prompt understanding.

Closed model: A proprietary model where weights are unavailable, like GPT-5, Claude Sonnet, or Google Gemini.

Cold start: The time from scaling a replica from zero to its first successful response (steps include GPU provisioning, container startup, model load, inference engine compilation).

ComfyUI: A workflow tool for assembling image pipelines (base model, refiner, LoRAs, ControlNets). Encourages modular, swappable components.

Compute-bound: An algorithm limited by available FLOPS rather than memory bandwidth. LLM prefill and image/video generation are usually compute bound.

Context Parallelism (CP): Replicates weights across GPUs and partitions the attention context. Essential for video models where attention works across a massive latent space.

Context window: The maximum number of tokens that a model can process across input, reasoning, and output for a single request.

Continuous batching (in-flight): Token-level interleaving of requests so GPU slots are always utilized. Minimizes per-user latency penalties of batching.

Control plane (multi-cloud): Global orchestrator for deploying models and allocating resources.

Core (CUDA): A general-purpose arithmetic unit that executes a wide range of scalar and element-wise operations.

Core (Tensor): A specialized hardware unit optimized for mixed-precision matrix multiply-accumulate (MMA) operations. Tensor Cores are the most important type of compute for inference.

Cross-attention: Conditioning one sequence (Q) on another's K/V (e.g., text conditioning images). Common in multimodal and denoising pipelines.

cuBLAS: CUDA's BLAS implementation offering high-quality GEMM and related primitives.

CUDA: NVIDIA's programming model and platform for GPU kernels, graphs, memory, and execution.

CUDA driver: A low-level interface between the application and the GPU hardware to manage memory and execution.

CUDA graph: A directed acyclic graph (DAG) of kernels and other GPU operations for optimizing repeated workflows.

CUDA kernel: A user-defined function that executes parallelized code on the GPU.

CUDA runtime: A developer-facing API for launching kernels and managing memory.

cuDNN: Primitives for building deep neural networks in CUDA.

CuTe: A domain-specific C++ template library that abstracts tiled tensor operations to help developers compose precision-aware, hardware-optimized GEMM and fused kernels.

CUTLASS: A CUDA C++ template library that provides building blocks for writing high-performance, architecture-tuned GEMM and related kernels.

Data sovereignty: Legal constraints around where model inputs and outputs are processed and stored geographically.

Decode: The memory-bound phase of LLM inference where the autoregressive generation loop emits one token per forward pass.

DeepGEMM: A library of clean and efficient GEMM kernels created by the DeepSeek AI team with strong performance in FP8.

Denoising model: The heart of diffusion pipelines that iteratively refines latent noise into an image or video.

Diarization: Segmenting audio by speaker ("who spoke when"); often paired with VAD in ASR pipelines.

Diffusers (library): Reference implementations for image and video generation pipelines.

Disaggregation: Separating prefill and decode onto independently scaling engines running on separate hardware resources.

Distillation: Training a smaller student to emulate a larger teacher model based on probability distributions, not just outputs, to retain model behavior on fewer parameters.

Docker: Containerization technology for building standardized packages of inference services with their dependencies.

Dockerfile: A human-readable file with well-specified, machine-interpretable instructions for creating an image.

Dynamic batching: Dynamic batching starts a batch when the batch is full or a short timer elapses, whichever comes first. Balances latency stability with utilization; superseded by continuous batching for LLMs.

Dynamic range (quantization): The range of absolute values that can be represented in a number format. Floating-point numbers have a higher dynamic range than integers with the same number of bytes thanks to their exponent-mantissa structure.

EAGLE (speculation): A small, purpose-built draft model trained to consume hidden states and propose multiple tokens for high acceptance rates in speculative decoding.

Elo (quality meta-metric): Head-to-head win-rate style scoring to compare model quality. Useful directional signal beyond intelligence benchmarks.

Embedding model: Encodes text or image input into fixed-dimensional vectors for semantic similarity, used in RAG and agent memory. Modern variants often use LLM backbones and Matryoshka representations.

Encoder: Network that converts raw inputs into internal representations (e.g., audio features in Whisper). Paired with a decoder in encoder-decoder models.

Evals: Task-specific tests that mirror real-world use cases for a model, used for product-specific model intelligence testing.

Expert Parallelism (EP): Shards experts of an MoE across GPUs; each GPU contains multiple full experts. Increases total throughput with low inter-GPU communication overhead.

Few-step image generation: Models that produce usable images in eight or fewer steps. Eighty to ninety percent faster but with noticeable quality tradeoffs; strong fit for real-time applications.

Feynman (architecture): A future NVIDIA generation after Rubin. Details are limited; expect continued emphasis on low-precision and memory bandwidth.

Fine-tuning: Adapts a pretrained base to a domain, often enabling much smaller models to meet quality needs.

FlashAttention: A series of optimized attention kernels that minimize memory traffic. FlashAttention 3 is written for Hopper, FlashAttention 4 targets Blackwell.

Floating-point data formats: Precisions like FP16, FP8, and FP4 used in inference with high dynamic range and an exponent-mantissa structure.

FLOPS: Floating-point operations per second, typically measured on Tensor Cores.

Foundation model: A model trained on broad data that serves as a base for multiple downstream tasks. Foundation models (e.g., GPT, Claude, Llama) are typically fine-tuned or used directly via prompting.

Function calling: Also known as tool calling or tool use, a model is given a set of available functions along with a prompt and returns a structured output including both selected functions and arguments for those functions.

GB200: An NVIDIA superchip that pairs a Grace CPU with a B200 GPU via high-bandwidth NVLink chip-to-chip connection. GB200s are used in rack-scale NVLink systems like the NVL72 and are useful for KV cache offloading, LoRA swapping, and other techniques that benefit from NVLink-C2C.

General matrix-matrix multiplication (GEMM): An algorithm in BLAS and the key operation for inference.

Generative AI: A class of models that, in contrast to predictive ML models, create new content across modalities (text, images, audio, video, code) by learning the underlying patterns of training data.

Generative Pretrained Transformer (GPT): A family of large language models for text generation created by OpenAI.

GH200: An NVIDIA superchip that pairs a Grace CPU with an H200 GPU via high-bandwidth NVLink chip-to-chip connection. GH200s are used in rack-scale NVLink systems like the NVL72 and are useful for KV cache offloading, LoRA swapping, and other techniques that benefit from NVLink-C2C.

Goodhart's Law: "When a measure becomes a target, it ceases to be a good measure."

GPU node: A standard chassis of 8 interconnected GPUs with NVLink and NVSwitch.

Grace CPU: ARM-based NVIDIA CPU with high-bandwidth chip-to-chip interconnects between the CPU and GPU. Used alongside Hopper and Blackwell GPUs.

Graphics Processing Unit (GPU): A highly parallel processor originally designed for graphics rendering and now widely used for training and inference of generative AI models.

gRPC: Structured, schema-first bidirectional streaming protocol.

Head (attention): One independent attention computation within a layer.

High-Bandwidth Memory (HBM): The memory used for VRAM on datacenter GPUs. Recent generations include HBM3, HBM3e, and HBM4.

Hopper (architecture): NVIDIA's 2022 GPU generation featuring FP8 support and async programming features.

Hyperscaler: Generalized cloud service providers like AWS and GCP.

Image generation pipeline: Foundation models for image generations are pipelines of multiple models: a text encoder, an iterative denoiser, and a VAE.

Inference: Serving AI models in production.

Inference engine: A high-performance runtime (vLLM, SGLang, TensorRT-LLM) with support for optimization techniques like batching, caching, quantization, and speculation.

InfiniBand: Inter-node interconnect for scaling inference and training across multiple nodes. While InfiniBand bandwidth is higher than alternatives like Ethernet, it is substantially lower than NVLink.

Input sequence: The tokens provided to a model as part of a request, processed during the prefill phase of inference.

Input Sequence Length (ISL): The number of tokens in the input sequence for a given request.

Instance (cloud): The provisioned virtual machine that includes GPU(s), CPU and RAM resources, storage, networking, and interconnect.

Integer data formats: Number formats like INT8 and INT4 with limited dynamic range.

Inter-token latency (ITL): Time between generated tokens during decode. Converts to perceived TPS (e.g., 2 milliseconds ITL equates to 500 TPS).

In-flight batching: See continuous batching. Token-level interleaving for high utilization with stable latency.

Iterative denoising (diffusion): Start from noise and progressively refine into an image or video in latent space.

Jitter traffic (bench): Adding randomness to arrival times and sequence shapes to more closely mirror real traffic than uniform or bursty synthetic loads.

Kernel fusion: Taking two or more kernels and re-implementing them into a single kernel that handles both operations, avoiding unnecessary round-trips through memory.

KV cache: Stored K/V tensors for each token to avoid recomputing attention, turning the attention equation from a quadratic-time to a linear-time operation.

L0/L1/L2 caches (GPU): On-chip cache memory hierarchy for instructions, shared memory, and global cache.

Large Language Model (LLM): A type of generative AI model that takes a text prompt and returns a new sequence of text. Many famous generative AI model families, including GPT, Claude, Llama, and DeepSeek, are LLMs.

Latency percentiles: Measuring latency on a percentile basis (P50/P90/P95/P99) for awareness of both the average and the worst-case user experience.

Latent consistency: A few-step strategy that predicts target latents directly, possibly repeated for refinement. Very fast; lower fidelity than full diffusion.

Latent space (images/videos): Lower-dimensional representation where denoising occurs (e.g., 128×128).

LLM: Large language model (e.g., GPT-5, Llama, DeepSeek).

Load testing: Sending sustained high traffic to probe throughput limits, queue behavior, and autoscaling.

Local (edge) inference: Running inference on end-user devices like phones and computers.

Logit biasing: Nudging or constraining token probabilities to steer structured outputs (e.g., JSON/tool calls). Applied post-logits before sampling.

Logits: A vector of non-normalized probabilities, one per token in the model's vocabulary, generated in each forward pass during decode.

Lookahead decoding: Constructs n-grams during inference to enable draft token prediction without a separate model.

LoRA: Low-rank adaptation, a lightweight fine-tuning method that produces small changes to models. Inference services often need to swap between thousands of LoRAs for a single foundation model.

Machine learning (ML): Predictive modeling for tasks like classification and trend forecasting, as opposed to generative AI which creates novel outputs.

Matmul: Matrix multiplication.

Matryoshka representations (embeddings): Nested vector schemes allowing variable dimensionality where the early part of the vector encodes more semantic meaning. Allows tradeoffs between vector size and embedding quality.

Medusa (speculation): Adds extra decoder heads via fine-tuning to generate multiple draft tokens per pass.

Microscaling formats: Floating-point data formats like MXFP8, MXFP4, and NVFP4 that use blockwise quantization with small-block scale factors (e.g., every 32 elements) to improve accuracy.

Mixture of Experts (MoE): A model architecture where linear layers of weights are separated into sparse experts. A router activates a subset of experts for each forward pass.

Model parallelism (overview): Splitting work across GPUs via Tensor, Expert, or Pipeline Parallelism. Parallelism strategy depends on model size, topology, and latency versus throughput goals.

Multi-cloud capacity management: A global scheduler placing workloads across providers and regions.

Multi-Instance GPU (MIG): A capability in larger Ampere, Hopper, Blackwell, and Rubin GPUs where the GPU can be carved into up to eight slices of memory and seven slices of compute.

Multi-node inference: Scaling across two or mode nodes using InfiniBand when one node of eight GPUs doesn't have enough VRAM for weights, activations, and KV cache. Requires appropriate parallelism strategies, either PP or EP between as TP uses too much all-to-all communication for InfiniBand.

Neocloud: Specialized cloud service providers focused on GPUs like Coreweave and Nebius.

Neural audio codec: A learned encoder that compresses audio into tokens and paired decoder that turns tokens back into audio.

NIM: A pre-packaged, containerized microservice for a specific model created by NVIDIA.

Node: The physical 8-GPU base unit with NVLink/NVSwitch. Multi-node adds InfiniBand between nodes.

NVFP4: NVIDIA's 4-bit floating-point microscaling number format with dual scale factors and blockwise quantization with a block size of 16.

NVIDIA Dynamo: An open-source distributed serving platform for KV reuse, disaggregation, and multi-GPU/multi-node orchestration.

NVL72: A rack-scale Blackwell system interconnecting 72 GPUs and 36 CPUs. Purpose-built for serving very large models with extreme throughput.

NVLink: A one-to-one communication layer between GPUs, up to 1800 GB/s on Blackwell and 900 GB/s on Hopper.

NVSwitch: An all-to-all communication layer on top of NVLink for coordination among all GPUs in a node.

N-gram speculation: Uses observed n-grams from prefill to propose long draft sequences during decode. Extremely effective for code completion.

Offline inference: Asynchronous batch processing of large jobs, optimized for throughput and cost over per-request latency.

Omni-modal: Models that accept multiple modalities of inputs (text, images, video, audio) and produce multiple modalities of output.

Online inference: Real-time serving of requests, optimized for tight latency budgets.

ONNX: An intermediate representation and runtime for models.

Open model: A model whose weights are freely available, like Llama, DeepSeek, or Whisper.

Ops:byte ratio (GPU): Peak operations per byte of memory bandwidth for a GPU at a given precision. Compare with arithmetic intensity to diagnose bottlenecks.

Output sequence: The tokens generated by a model during the decode phase of inference.

Output Sequence Length (OSL): The number of tokens in the output sequence generated by a model for a given request.

Out-of-memory error (OOM): A common failure where the GPU runs out of VRAM to load weights or execute inference.

PagedAttention: An optimization for attention where KV blocks are stored in fixed-size pages to improve performance, especially with long context.

PCIe (GPU form factor): A form factor for datacenter GPUs that uses standard PCI express slots for connection. PCIe GPUs often have lower base specs and fewer interconnect options than SXM variants of the same GPU.

Perceived TPS: Tokens per second observed by a single user during streaming output. This latency metric is a more specific term for what people usually mean when they say TPS.

Pipeline Parallelism (PP): Splits layers into stages across GPUs. While acceptable for multi-node with dense models; PP introduces bubbles in the pipeline where some GPUs are idle while waiting for other steps to finish.

Prefill: The compute-bound phase of LLM inference where the input sequence is processed and the KV cache is built.

Prefix caching: Reuses KV for shared prefixes across requests to skip prefill. Majorly improves TTFT for code completion, multi-turn chat, and agents.

Pretraining: Large-scale (usually self-supervised) training on broad corpora to create a base model.

Prompt: The instruction to the model; for diffusion also includes a negative prompt and step/guidance parameters.

PyTorch compile (torch.compile): Graph capture and kernel selection/fusion targeting a specific GPU. Cache compiled engines to cut cold-start times.

PyTorch Profiler: A developer tool measuring CPU and GPU time and memory per operation.

Quantization (post-training): Lowering precision of weights, activations, and potentially KV cache to reduce compute and memory bandwidth demands.

Quantization-aware training: A training technique in which quantization scales are computed and weights are optimized jointly so that the final model is already calibrated for low-precision deployment.

Queue (request): Holds excess traffic while autoscaling brings replicas online.

Real-time factor (RTF): A measurement of how quickly ASR models can transcribe audio. Transcribing an hour of audio in six seconds is an RTF of 600.

Retrieval-augmented generation (RAG): A common application pattern that fetches additional context for the LLM beyond the prompt.

Ring attention: A Context Parallelism mechanism in which GPUs pass partial attention results in a ring. Reduces all-to-all pressure for very large contexts.

Roofline model: Plots arithmetic intensity with bandwidth and compute ceilings, creating a visual guide on whether to optimize memory or compute.

Rotary positional embeddings (RoPE): A positional encoding scheme that encodes positions as learned rotations, improving long-context extrapolation at the cost of higher memory demands for attention during inference.

Routing (inference): Placing requests on replicas based on load, KV cache, available LoRAs, and sequence shapes to improve speed and utilization.

Rubin (architecture): Next NVIDIA generation (2026) introducing HBM4 and CPX for compute-bound workloads.

Sampling (decode): The process of selecting an output token based on the generated logits. Common strategies include greedy (argmax), temperature-based sampling, top-k, and top-p (nucleus) sampling.

Scale factor (quantization): Multipliers used to map low-precision values to their original number formats.

Scale to zero: Turn off all replicas when idle; spin up on demand. Requires fast cold starts and robust queueing; best for predictable or dev workloads.

SDXL: An instructive, earlier diffusion image pipeline (base + refiner + CLIP). Modern systems retain the structure with larger, more capable components.

Service Level Agreement (SLA): A contractual promise of latency, throughput, uptime, or other performance factor from a system.

Service Level Objective (SLO): An internal target designed to meet or beat the SLA for a given system.

SGLang: A fast inference engine with flexible frontend/backends and strong MoE support.

Shadow traffic: Mirroring real production requests to a candidate deployment.

SNAC (audio decoder): A performant audio decoder path often paired with TTS token streams.

Softmax: Converts scores to probabilities in attention and normalizes logits to a probability distribution in decoding.

Sparsity (FLOPS): In tensors with 2:4 structured sparsity, where 50 percent of the values are 0, Tensor Cores can skip multiplication by 0. Most inference is dense, not sparse.

Special Function Unit (SFU): A dedicated hardware unit that accelerates specific math operations like sine and cosine, keeping specialized operations off of CUDA Cores.

Speculative decoding: A family of strategies for generating and validating draft tokens to generate multiple tokens per forward pass during decode.

Streaming Multiprocessor (SM): GPU compute unit containing cores and cache.

Structured output: LLM output that adheres to a specific schema. Created by constraining generation to a supplied schema via logit biasing rather than via prompting.

SXM (GPU form factor): A socketed GPU module that supports higher-bandwidth connections and delivers more power than PCIe. SXM form factor GPUs often have higher base specs and are the standard for inference.

Temperature: Controls randomness in token selection: lower values (e.g., 0.1) make output more deterministic; higher values (e.g., 1.5) increase diversity.

Tensor Parallelism (TP): Splits tensor operations across GPUs within a node. Best per-user latency; requires frequent all-reduce synchronization.

TensorRT: NVIDIA's optimized runtime for high-performance inference with fused kernels, quantization, and other optimizations.

TensorRT-LLM: An inference engine built by NVIDIA that provides a Python API and both TensorRT-engine and PyTorch-backend execution paths with fused kernels, quantization, and speculative decoding.

Thread: The minimal execution unit on a GPU. Kernels launch many threads to achieve massive parallelism.

Throughput: Total work per unit time (e.g., total tokens per second).

Time to first byte (TTFB): Time until first byte of output is returned, a latency metric.

Time to first token (TTFT): Time until first token of output is returned, a latency metric.

Token: The atomic unit of text processing in LLMs. A token is an integer that represents a string of characters. In English, there is approximately a 4:3 token:word ratio for most tokenizers.

Tokenizer: Deterministically converts strings into sequences of tokens, and vise versa. Models have different tokenizers, and more efficient tokenizers improve end-to-end latency.

Tokens per second (TPS): See perceived TPS. A latency metric for the number of tokens streamed to the end user per second.

Training: The process of learning model weights from data using back-propagation and optimization. Training is compute-intensive, typically runs on large GPU clusters, and produces the weights used in inference.

Transformer: The foundational architecture behind generative AI models.

Transformers (library): Reference implementations for LLMs and other transformers-based models.

Triton Inference Server: A production serving framework by NVIDIA with support for multiple backends.

VAE (variational autoencoder): Used in inference to decode from latent space to pixel space for image and video generation models (also used for encoding from pixel to latent space during training).

Vector database: A database for storing and querying the semantic vectors created by embedding models.

Vector similarity: A check between two vectors to see how close together they are based on an equation like cosine similarity. Vectors with high similarity encode similar semantic meaning.

Vera CPU: ARM-based NVIDIA CPU with high-bandwidth chip-to-chip interconnects between the CPU and GPU. Succeeds Grace GPUs alongside the Rubin GPU architecture generation.

Vision-language model (VLM): Accepts images and video plus text prompts and outputs text.

vLLM: A widely adopted inference engine with broad model and hardware support and strong defaults.

Vocabulary: The total set of tokens, usually more than 100,000, that an LLM uses to represent data.

Voice activity detection (VAD): A lightweight model that segments streams/files into speech-containing chunks for ASR.

VRAM (device memory): On-GPU memory used for weights, KV, and activations. Total VRAM gates model size and KV headroom; bandwidth gates decode TPS.

WebSocket: Lightweight, bidirectional streaming transport. Ideal for unstructured audio chunks and real-time UX.

Weights-only quantization: Reduces precision for model weights in linear layers while preserving other model components like KV cache and attention at higher precision. A conservative approach to quantization with the best quality preservation but the lowest performance improvements.

Workload plane (multi-cloud): An individual cluster with compute resources that runs inference and processes requests.

APPENDIX B

Recommended Reading

This book is an introduction to the field of inference engineering. There is endless depth to explore in every one of the technologies and techniques behind performant inference at scale.

If you're in the market for another book to continue learning, I have three recommendations:

- *AI Engineering: Building Applications with Foundation Models* by Chip Huyen (O'Reilly Media, 2025): This incredibly popular book introduces the full breadth of AI engineering topics.
- *Build a Large Language Model (From Scratch)* by Sebastian Raschka (Manning, 2024): This hands-on book provides a detailed look at LLM architecture.
- *AI Systems Performance Engineering: Optimizing Model Training and Inference Workloads with GPUs, CUDA, and PyTorch* by Chris Fregly (O'Reilly Media, 2025): This brand-new book focuses on building for performance.

The AI industry moves fast, and new models, research, and implementations are constantly being released. My colleagues and I publish our latest work on the Baseten blog, which you can access at https://www.baseten.com/blog.

This appendix provides a list of papers, documentation, books, and blogs to further support your next steps as an inference engineer. Resources are organized by topic and alphabetized by title within each section.

Architecture

"Attention is All You Need," by Ashish Vaswani et al. (*Neural Information Processing Systems*, 2017), https://arxiv.org/abs/1706.03762



"BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," by Jacob Devlin et al. (*North American Chapter of the Association for Computational Linguistics, 2019),* https://arxiv.org/abs/1810.04805



"BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models," by Junnan Li et al. (International Conference on Machine Learning, 2023), https://arxiv.org/abs/2301.12597



Deep Learning, by Ian Goodfellow, Yoshua Bengio, and Aaron Courville (The MIT Press, 2016), https://www.deeplearningbook.org/



Deep Learning with Python (2nd Edition), by François Chollet (Manning, 2021), https://www.manning.com/books/deep-learning-with-python-second-edition



"Denoising Diffusion Probabilistic Models," by Jonathan Ho et al. (*ArXiv* abs/2006.11239, 2020), https://arxiv.org/abs/2006.11239



"DiT: Scalable Diffusion Models with Transformers," by William Peebles and Saining Xie *(International Conference on Computer Vision (ICCV)*, 2022), https://arxiv.org/abs/2212.09748



"FlashAttention: Fast and Memory-Efficient Exact Attention with IO Awareness," by Tri Dao et al. (*ArXiv* abs/2205.14135, 2022), https://arxiv.org/abs/2205.14135



"FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning," by Tri Dao (*ArXiv* abs/2307.08691, 2023), https://arxiv.org/abs/2307.08691



"FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision," by Jay Shah et al. (*ArXiv* abs/2407.08608, 2024), https://arxiv.org/abs/2407.08608



Flash-Attention-4, by Tri Dao (Dao AI Research Lab, 2025), https://github.com/Dao-AILab/flash-attention



"Imagen Video: High Definition Video Generation with Diffusion Models," by Jonathan Ho et al. (*ArXiv* abs/2210.02303, 2022), https://arxiv.org/abs/2210.02303



"Language Models Are Few-Shot Learners," by Tom Brown et al. (*ArXiv* abs/2005.14165), https://arxiv.org/abs/2005.14165



"Learning Transferable Visual Models from Natural Language Supervision," by Alec Radford et al. (*International Conference on Machine Learning*, 2021), https://arxiv.org/abs/2103.00020



"Longformer: The Long-Document Transformer," by Iz Beltagy et al. (*ArXiv* abs/2004.05150, 2020), https://arxiv.org/abs/2004.05150



"Mamba: Linear-Time Sequence Modeling with Selective State Spaces," by Albert Gu and Tri Dao (*ArXiv* abs/2312.00752, 2023), https://arxiv.org/abs/2312.00752



"Matryoshka Representation Learning," by Aditya Kusupati et al. (*Neural Information Processing Systems*, 2022), https://arxiv.org/abs/2205.13147



"Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer," by Noam Shazeer et al. (*ArXiv* abs/1701.06538, 2017), https://arxiv.org/abs/1701.06538



"Reformer: The Efficient Transformer," by Nikita Kitaev et al. (*ArXiv* abs/2001.04451, 2020), https://arxiv.org/abs/2001.04451



"Robust Speech Recognition via Large-Scale Weak Supervision," by Alec Radford et al. (*International Conference on Machine Learning*, 2022), https://arxiv.org/abs/2212.04356



"RoFormer: Enhanced Transformer with Rotary Position Embedding," by Jianlin Su et al. (*ArXiv* abs/2104.09864, 2021), https://arxiv.org/abs/2104.09864



"SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis," by Dustin Podell et al. (*ArXiv* abs/2307.01952, 2023), https://arxiv.org/abs/2307.01952



"Segment Anything," by Alexander Kirillov et al. (*2023 IEEE/CVF International Conference on Computer Vision (ICCV),* 2023), https://arxiv.org/abs/2304.02643



"Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks," by Nils Reimers and Iryna Gurevych (*ArXiv* abs/1908.10084, 2019), https://arxiv.org/abs/1908.10084



"The Llama 3 Herd of Models," by Aaron Grattafiori et al. (*ArXiv* 2407.21783, 2024), https://arxiv.org/abs/2407.21783



"Video Diffusion Models," by Jonathan Ho et al. (*ArXiv* abs/2204.03458, 2022), https://arxiv.org/abs/2204.03458



"Visual Instruction Tuning," by Haotian Liu et al. (*ArXiv* abs/2304.08485, 2023), https://arxiv.org/abs/2304.08485



Developer Tools

BitsAndBytes by bitsandbytes-foundation, https://github.com/TimDettmers/bitsandbytes



ComfyUI by comfyanonymous, https://github.com/comfyanonymous/ComfyUI



"CUDA by Example: An Introduction to General Purpose GPU Programming," by Jason Sanders and Edward Kandrot (NVIDIA developer, 2025), https://developer.nvidia.com/cuda-example



"CUDA C++ Programming Guide Release 13.0," by NVIDIA (2025), https://docs.nvidia.com/cuda/cuda-c-programming-guide/



"CUDA cuBLAS Release 13.0," by NVIDIA (2025), https://docs.nvidia.com/cuda/cublas/



CUTLASS by NVIDIA, https://github.com/NVIDIA/cutlass



DeepGEMM by DeepSeek-ai, https://github.com/deepseek-ai/DeepGEMM



Hugging Face Diffusers by Hugging Face, https://huggingface.co/docs/diffusers/index



LMCache by LMCache Project, https://github.com/LMCache/LMCache



“NVIDIA Dynamo Documentation,” by NVIDIA (2025), https://docs.nvidia.com/dynamo/latest/index.html



NVIDIA Nsight Systems by NVIDIA, https://developer.nvidia.com/nsight-systems



NVIDIA Triton Inference Server by NVIDIA, https://github.com/triton-inference-server/server



ONNX Runtime by Microsoft, https://onnxruntime.ai/



"PyTorch Performance Tuning Guide," by Szymon Migacz (PyTorch Foundation, 2020), https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html



"PyTorch Profiler," by Shivam Raikundalia (PyTorch Foundation, 2021), https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html



SGLang Project by LMSYS Org, https://github.com/sgl-project/sglang



"TensorRT Documentation," by NVIDIA (2025), https://docs.nvidia.com/deeplearning/tensorrt/



TensorRT-LLM by NVIDIA, https://github.com/NVIDIA/TensorRT-LLM



Transformers by Hugging Face, https://huggingface.co/docs/transformers/index



vLLM Project by The Linux Foundation, https://github.com/vllm-project/vllm



Frontier Open Models

DeepSeek, by DeepSeek AI, https://huggingface.co/deepseek-ai



FLUX, by Black Forest Labs**,** https://huggingface.co/black-forest-labs



Gemma, by Google, https://huggingface.co/google



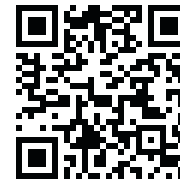
GLM, by Z.ai, https://huggingface.co/zai-org



GPT OSS, by OpenAI, https://huggingface.co/openai



Kimi, by Moonshot AI, https://huggingface.co/moonshotai



Llama, by Meta Llama, https://huggingface.co/meta-llama



MiniMax, by MiniMax AI, https://huggingface.co/MiniMaxAI



Mistral, by Mistral AI, https://huggingface.co/mistralai



Nemotron, by NVIDIA, https://huggingface.co/nvidia



Orpheus, by Canopy Labs, https://huggingface.co/canopylabs



Qwen, by Alibaba Qwen, https://huggingface.co/Qwen



Wan, by Wan-AI,
https://huggingface.co/Wan-AI



Whisper, by OpenAI, https://huggingface.co/openai



GPU Infrastructure

Designing Data-Intensive Applications, by Martin Kleppmann (O'Reilly Media, 2017) https://dataintensive.net/



Grace Hopper / Grace Blackwell Systems by NVIDIA, https://www.nvidia.com/en-us/data-center/grace-cpu/



GPU Glossary, by Frye et al. (Modal, 2025), https://modal.com/gpu-glossary



InfiniBand, by NVIDIA, https://www.nvidia.com/en-us/networking/products/infiniband/



Kubernetes Documentation, by The Kubernetes Authors (The Linux Foundation, 2025), https://kubernetes.io/docs/home/



NVIDIA Blackwell Architecture Technical Brief: Built for the Age of AI Reasoning, (NVIDIA, 2025), https://resources.nvidia.com/en-us-blackwell-architecture?ncid=no-ncid



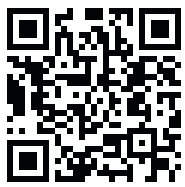
NVIDIA H100 Tensor Core GPU Architecture: Exceptional Performance, Scalability and Security for the Data Center, (NVIDIA, 2023), https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c



"NVIDIA Tesla: A Unified Graphics and Computing Architecture," by E. Lindholm et al. (*IEEE Micro*, March–April 2008), https://ieeexplore.ieee.org/document/4523358



NVLink / *NVSwitch,* by NVIDIA, https://www.nvidia.com/en-us/data-center/nvlink/



Programming Massively Parallel Processors: A Hands-on Approach, by Wen-mei Hwu, David Kirk, Izzat El Hajj (Morgan Kaufmann, 2022), https://www.goodreads.com/work/editions/10244675-programming-massively-parallel-processors-a-hands-on-approach



SemiAnalysis, by Dylan Patel (SemiAnalysis, 2025), https://semianalysis.com/



Site Reliability Engineering: How Google Runs Production Services, edited by Betsy Beyer et al. (O'Reilly Media, 2017), https://sre.google/books/



Inference Optimization Research

"Adversarial Diffusion Distillation," by Axel Sauer et al. (*European Conference on Computer Vision*, 2023), https://arxiv.org/abs/2311.17042



"Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation," by Ofir Press, Noah Smith, and Mike Lewis (*ArXiv* abs/2108.12409, 2021), https://arxiv.org/abs/2108.12409



"AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration," by Song Han (MIT, 2024), https://github.com/mit-han-lab/llm-awq



Cache-DIT by Vipshop, https://github.com/vipshop/cache-dit



"CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion," by Jiayi Yao et al. (*Proceedings of the Twentieth European Conference on Computer Systems*, 2024), https://arxiv.org/abs/2405.16444



"Adding Conditional Control to Text-to-Image Diffusion Models," by Lymin Zhang et al. (*International Conference on Computer Vision,* 2023), https://arxiv.org/abs/2302.05543



"Beyond the Buzz: A Pragmatic Take on Inference Disaggregation," by Tiyasa Mitra et al. (*ArXiv* abs/2506.05508, 2025), https://arxiv.org/abs/2506.05508



"Break the Sequential Dependency of LLM Inference Using Lookahead Decoding," by Yichao Fu et al. (*ArXiv* abs/2402.02057, 2024), https://arxiv.org/abs/2402.02057



"EAGLE: Speculative Sampling Requires Rethinking Feature Uncertainty," by Yuhui Li et al. (*ArXiv* abs/2401.15077, 2024), https://arxiv.org/abs/2401.15077



"EAGLE-2: Faster Inference of Language Models with Dynamic Draft Trees," by Yuhui Li et al. (*Conference on Empirical Methods in Natural Language Processing*, 2024), https://arxiv.org/abs/2406.16858



"EAGLE-3: Scaling up Inference Acceleration of Large Language Models via Training-Time Test," by Yuhui Li et al. (*ArXiv* abs/2503.01840, 2025), https://arxiv.org/abs/2503.01840



"FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving," by Ye et al. (*ArXiv* abs/2501.01005, 2025), https://arxiv.org/abs/2501.01005



"GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers," by Elias Frantar (*ArXiv* abs/2210.17323, 2022), https://arxiv.org/abs/2210.17323



"High-Resolution Image Synthesis with Latent Diffusion Models," by Robin Rombach et al. (*Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021), https://arxiv.org/abs/2112.10752



"Latent Consistency Models: Synthesizing High-Resolution Images with Few-Step Inference," by Simian Luo (*ArXiv* abs/2310.04378, 2023), https://arxiv.org/abs/2310.04378



"LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale," by Tim Dettmers et al. (*ArXiv* abs/2208.07339, 2022), https://arxiv.org/abs/2208.07339



"Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads," by Tianle Cai et al. (*ArXiv* abs/2401.10774, 2024), https://arxiv.org/abs/2401.10774



"Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," by Mohammad Shoeybi et al. (*ArXiv* abs/1909.08053, 2019), https://arxiv.org/abs/1909.08053



"Efficient Memory Management for Large Language Model Serving with PagedAttention," by Woosuk Kwon et al. (*Proceedings of the 29th Symposium on Operating Systems Principles*, 2023), https://arxiv.org/abs/2309.06180



"Fast Inference from Transformers via Speculative Decoding," by Yaniv Leviathan et al. (*International Conference on Machine Learning*, 2022), https://arxiv.org/abs/2211.17192



"Ring Attention with Blockwise Transformers for Near-Infinite Context," by Hao Lin et al. (*ArXiv* abs/2310.01889, 2023), https://arxiv.org/abs/2310.01889



"SageAttention: Accurate 8-Bit Attention for Plug-and-play Inference Acceleration," by Jintao Zhang et al. (*ArXiv* abs/2410.02367, 2024), https://arxiv.org/abs/2410.02367



Sequence/Context Parallelism, by Megatron-LM for NVIDIA https://github.com/NVIDIA/Megatron-LM



SmoothQuant by Song Han (MIT), https://github.com/mit-han-lab/smoothquant



"SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot," by Elias Frantar and Dan Alistarh, (*ArXiv* abs/2301.00774, 2023), https://arxiv.org/abs/2301.00774



"SpecVLM: Fast Speculative Decoding in Vision-Language Models," Haiduo Huang et al. (*ArXiv* abs/2509.11815, 2025), https://arxiv.org/abs/2509.11815



"TeaCache: Efficient KV Cache Compression via Tensor Decomposition," by Feng Lu et al. (Alibaba TongYi Vision Intelligence Lab, *ArXiv* abs/2411.19108, 2025), https://github.com/ali-vilab/TeaCache



Intelligence Evaluation

ARC AGI Prize by Greg Kamradt (2025), https://arcprize.org/



Evals for AI Engineers: Systematically Measuring and Improving AI Applications, by Shreya Shankar and Hamel Husain (O'Reilly Media, forthcoming 2026) https://www.oreilly.com/library/view/evals-for-ai/9798341660717/



Grade School Math: Training Verifiers to Solve Math Word Problems, by Karl Cobbe and Vineet Kosaraju (*ArXiv* abs/2110.14168, 2021), https://github.com/openai/grade-school-math



"How to Fine-Tune Qwen3 to GPT-4o Level Performance," by Greg Schoeninger (*Fine-Tune Fridays*, Oxen AI, 2025), https://ghost.oxen.ai/how-to-fine-tune-qwen3-to-gpt-4o-level-performance/



"Humanity's Last Exam," by Long Phan et al. (*Center for AI Safety and Scale AI, ArXiv* abs/2501.14249, 2025), https://www.lastexam.ai/



"HumanEval: Evaluating Large Language Models Trained on Code," by Michelle Pokrass, Qiming Yuan, and Yichen Xu (OpenAI, 2021), https://github.com/openai/human-eval



“MMLU: Measuring Massive Multitask Language Understanding,” by Dan Hendrycks et al. (*Proceedings of the International Conference on Learning Representations (ICLR)*, 2021), https://github.com/hendrycks/test



“MTEB: Massive Text Embedding Benchmark,” by Niklas Muenninghoff et al. (*Conference of the European Chapter of the Association for Computational Linguistics*, 2022), https://arxiv.org/abs/2210.07316



“SWE-Bench: Can Language Models Resolve Real-World Github Issues?” by Carlos Jimenez et al. (*Proceedings of the International Conference on Learning Representations (ICLR)*, 2024), https://www.swebench.com/



Acknowledgements

This book was made possible by people who invested in me early, without assurance of success. My thanks to Mike Bilodeau and Dannie Herzberg, who greenlit this project when I was supposed to be working on a dozen other priorities. More thanks to Tuhin Srivastava, Amir Haghighat, Phil Howes, Pankaj Gupta, and Emmiliese von Avis, who hired me four years ago, even though I had no formal experience in the role. To Megha, for everything. And to my family, who has bet on me for my entire life.

Thank you to the many reviewers who helped me make this book as accurate and precise as possible: Raymond Cano, Tianshu Cheng, Tri Dao, Michael Feil, William Gao, Pankaj Gupta, Amir Haghighat, Mahmoud Hassan, Nidhi Hiremath, Phil Howes, Tyron Jung, Kaz Kato, Madison Kanna, Alex Ker, Harry Kim, Colin McGrath, Deepak Nagaraj, Eskil Olsen, Ed Shrager, Aaryam Sharma, and Joey Zwicker. Any errors that remain are my sole responsibility.

Thank you to Robin Bourjaily for editing this book, along with just about everything else I've ever written.

Thank you to Luke de Hass for the cover and internal illustrations, to Aaron Relph for design guidance, to Javier Crocco for internal illustrations, and to Raúl López for additional design assistance.

Thank you to the many researchers behind the open models, contributors to the open-source libraries, and authors of the papers mentioned in this book. You are the giants upon whose shoulders we stand.

And finally, my thanks to you, the reader, without whom I would merely be creating very elaborate training data.