Author Name

Self-Hosted AI Inference

A Systems Engineer’s Guide

December 14, 2025

Springer Nature

Contents

Chapter 1
Introduction to Self-Hosted Inference

Abstract This chapter introduces self-hosted AI inference: running models on infrastructure you control rather than consuming managed API services. We cover the fundamental differences between training and inference, the economic and strategic considerations for controlling your inference stack, and provide a roadmap for what you'll build throughout this book. By the end of this chapter, you'll run your first model and make your first inference request.

1.1 The Rise of Self-Hosted AI

The landscape of AI deployment shifted fundamentally in 2023 when Meta released Llama 2 under a permissive license [1]. For the first time, a model competitive with commercial APIs was available for independent deployment—on your own terms, on infrastructure you control. Other organizations followed: Mistral AI released their 7B model in September 2023 [2], Alibaba open-sourced the Qwen family [3], Microsoft published the Phi series optimized for efficiency [4], and Google contributed Gemma [5].

These models have modest compute requirements. A 7 billion parameter model quantized to 4-bit precision requires approximately 4GB of VRAM—achievable on a laptop GPU, a cloud instance, or a MacBook's unified memory. The barrier to entry is no longer access to models or specialized infrastructure; it's the systems knowledge required to deploy and operate them reliably.

When you use a managed API, the trade-offs are straightforward: you pay per token, accept the provider's latency (typically 50–200ms for first token), and send your data to external servers. When you control the inference stack, those trade-offs invert. Your per-request costs become predictable. Latency becomes a function of your chosen infrastructure rather than shared services and network round-trips. Your data stays within boundaries you define.

The complexity trade-off is real. Running your own inference requires understanding GPU memory hierarchies, model quantization formats, inference engine architectures, and production deployment patterns. This book provides that knowledge systematically, whether you're deploying on a cloud GPU instance, a colocated server, or a workstation under your desk.

1.2 Training vs. Inference: Understanding the Difference

The terms "training" and "inference" describe two fundamentally different computational processes. Training creates a model by learning patterns from data. Inference uses that trained model to generate outputs for new inputs. The resource requirements differ by orders of magnitude, which is why you can run inference on a laptop but training typically requires a data center.

This book focuses on inference, not training. We won't derive backpropagation equations or explain attention mechanisms mathematically—other texts cover that well [6, 7]. Instead, we treat models as artifacts you download and deploy. The explanations below provide enough context to understand resource requirements and make informed infrastructure decisions. If terms like "parameters" or "gradients" are unfamiliar, the brief descriptions here should suffice; deeper understanding isn't required to build and operate inference systems.

1.2.1 What Happens During Training

Training a large language model involves adjusting billions of numerical parameters so the model can predict the next token in a sequence. The process works in three steps, repeated millions of times:

1. **Forward pass**: Input text flows through the network, producing a prediction for the next token.
2. **Loss calculation**: The prediction is compared to the actual next token, producing an error signal.
3. **Backward pass**: The error propagates backward through the network via gradient descent, slightly adjusting each parameter to reduce future errors.

The backward pass is the expensive part. It requires storing intermediate activations from the forward pass (consuming memory) and computing gradients for every parameter (consuming compute). For a 7B parameter model, this means computing and storing 7 billion gradients per training step.

Training Llama 2 70B required 1.7 million GPU hours on A100 80GB GPUs [1]. At current cloud prices of approximately \$2/hour per A100, that represents over \$3 million in compute costs—before accounting for failed experiments, hyperparameter tuning, and infrastructure overhead. Training also requires massive datasets: Llama 2 used 2 trillion tokens of text data.

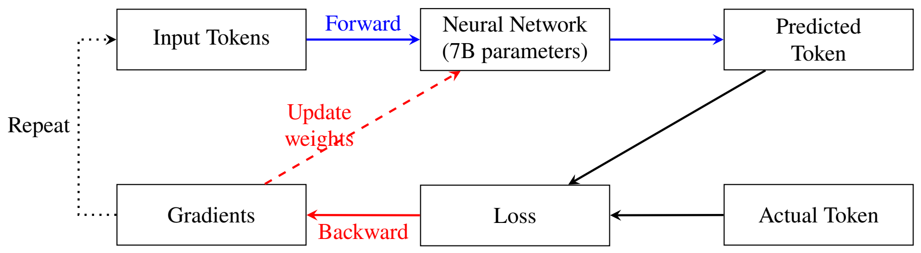

Fig. 1.1 The training loop. Each iteration performs a forward pass (blue) to generate predictions, computes the loss against actual targets, then propagates gradients backward (red) to update model weights. Training Llama 2 70B required approximately 1.7 million GPU hours of this loop.

1.2.2 What Happens During Inference

Inference is simpler: it runs only the forward pass. There's no loss calculation, no gradient computation, and no parameter updates. The model weights are fixed; you're simply using them to transform inputs into outputs.

For large language models, inference works autoregressively—the model generates one token at a time, feeding each generated token back as input to produce the next. Generating a 100-token response requires 100 forward passes through the network. This sequential dependency is why LLM inference can feel slow even on fast hardware: you cannot parallelize the generation of a single response.

The resource profile differs substantially from training:

- **Memory**: You need to store the model weights and a *key-value cache* (KV cache) that grows with context length. No gradient storage required.
- **Compute**: Forward passes only, with operations dominated by matrix multiplications between inputs and weights.
- **Data**: A single prompt, not terabytes of training data.

The KV cache deserves attention because it's often the limiting factor for long-context inference. During generation, the model caches intermediate computations (keys and values from the attention mechanism) to avoid redundant work. For a 7B model with a 4096-token context, the KV cache consumes approximately 1GB of memory at FP16 precision. At 128K context length, that grows to roughly 32GB—potentially exceeding the model weights themselves. Chapter **??** covers memory calculations in detail.

A single inference request on a 7B model takes 10–100 milliseconds for the first token (depending on prompt length) and 20–50 milliseconds per subsequent token on a modern consumer GPU. This is the timescale you'll work with throughout this book: milliseconds per token, not months per training run.

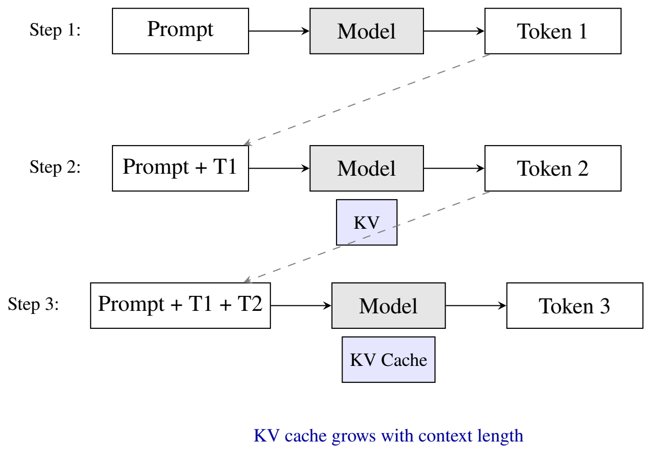

Fig. 1.2 Autoregressive inference. Each token is generated by a forward pass through the model, then appended to the context for the next iteration. The KV cache stores intermediate attention computations to avoid redundant work, growing with each generated token.

1.2.3 Resource Comparison

Table 1.1 summarizes the resource differences. The key insight: training is a one-time cost borne by model creators (Meta, Mistral, etc.), while inference is the ongoing cost you pay when deploying models. This book assumes someone else has already trained the model; your job is to run it efficiently.

Table 1.1 Resource requirements for training versus inference of a 70B parameter model.

Aspect	Training	Inference
Hardware	8+ A100 80GB GPUs	1–2 consumer GPUs
Time per run	Weeks to months	10–100 ms per token
Data required	Trillions of tokens	Single prompt
Memory usage	Weights + gradients + activations	Weights + KV cache
Compute cost	\$1–10 million	\$0.0001–0.01 per request
Who pays	Model creator	You (the deployer)

The implication is significant: you can leverage billions of dollars of training investment by downloading open-weight models and running inference on hardware you control. A \$1,000 GPU can serve the same model that cost millions to train.

1.3 Why Self-Host?

Organizations and individuals choose self-hosted inference for four primary reasons: cost control, data privacy, latency requirements, and customization needs. The weight of each factor varies by use case. A hobbyist experimenting on weekends has different priorities than a healthcare company processing patient records.

1.3.1 Cost Control

API pricing follows a per-token model. As of Dec 2025, OpenAI's GPT-5 costs $1.75 per million input tokens and $14.00 per million output tokens [8]. Anthropic's Claude Sonnet 4.5 costs $3.00 input and $15.00 output per million tokens, while Claude Opus 4.5 runs $5.00 input and $25.00 output [9]. These costs accumulate quickly at scale.

Consider a customer support application handling 10,000 conversations per day, averaging 500 input tokens and 500 output tokens per conversation. At GPT-5 rates, that's approximately $56 per day ($8.75 input + $70 output), or $2,340 per month. For a coding assistant processing 100,000 requests daily with longer outputs, monthly API costs can reach $40,000–70,000 depending on the model tier.

Self-hosted costs work differently. The primary expenses are:

- **Hardware**: A single RTX 4090 ($3,000) can run a 7B model at 30–50 tokens/second, or a quantized 70B model at 5–10 tokens/second.
- **Electricity**: GPUs under load draw 300–400W. At $0.12/kWh, that's roughly $25–35/month for continuous operation.
- **Maintenance**: Minimal for small deployments; significant for multi-GPU clusters.

The break-even calculation depends on utilization. A $2,000 GPU investment pays for itself in 1–2 months if you're replacing $1,500/month in API costs. If your usage is sporadic—a few hundred requests per day—APIs may remain more economical. Chapter **??** provides detailed cost models for different deployment scales.

1.3.2 Privacy and Data Sovereignty

When you send a prompt to an external API, that data travels over the internet to servers you don't control. Most providers retain prompts and responses for some period—OpenAI's default retention is 30 days, with options to disable [10]. Even with retention disabled, your data still traverses external infrastructure.

For many applications, this is acceptable. For others, it's a non-starter:

- **Healthcare**: Patient data falls under HIPAA in the US, requiring Business Associate Agreements (BAAs) with any third party processing protected health information. Not all API providers offer BAAs, and even those that do add compliance overhead.

- **Legal**: Attorney-client privilege may be compromised if confidential case details are processed by external services.
- **Finance**: Regulations like GDPR, SOX, and various banking laws impose strict requirements on where data can be processed and stored.
- **Government**: Many agencies require data to remain within specific geographic boundaries or on classified networks.

There are two paths to addressing these concerns. First, major cloud providers offer compliant AI services—AWS, Azure, and Google Cloud all provide HIPAA-eligible and SOC 2 certified infrastructure for running inference workloads. You can deploy open models on these platforms while satisfying regulatory requirements. Second, you can run inference on your own infrastructure, whether that's a server in your office or a private cloud deployment.

This book covers both approaches. "Self-hosted" doesn't mean running your own data center—it means running inference infrastructure you control, wherever that infrastructure lives. You might deploy on a HIPAA-compliant AWS instance, a GPU server in a colocation facility, or a workstation under your desk. The common thread is that you choose the model, control the deployment, and own the infrastructure decisions.

The trade-off is responsibility. You become accountable for securing the inference infrastructure, managing access controls, and maintaining audit logs. For air-gapped environments—networks with no internet connectivity—self-hosting is the only option. For compliant cloud deployments, you inherit some security controls from the provider while retaining others.

1.3.3 Latency and Performance

Managed API latency has two components: network round-trip time and inference time. A request from New York to a data center in California adds 60–80ms of network latency before inference even begins. For interactive applications—chat interfaces, code completion, real-time assistants—this overhead matters.

When you control the inference stack, you control placement. Deploy in the same region as your users, or on the same network as your application servers, and network latency drops to single-digit milliseconds. For latency-critical workloads, you can colocate inference with the consuming application.

Consistency is equally important. Managed APIs serve many customers on shared infrastructure. During peak demand, you compete for resources with other users, leading to variable response times. Your P99 latency might be 3x your median latency. With dedicated infrastructure—whether a reserved cloud instance or on-premise hardware—your latency distribution tightens because you're not sharing resources.

The performance trade-off depends on model size and hardware. A 7B model on a mid-range GPU delivers 30–50 tokens per second with time-to-first-token under 100ms. That's competitive with managed APIs for most workloads. Larger models on slower hardware may underperform managed services that run on cutting-edge GPUs with aggressive optimization. Chapter **??** helps you match hardware to performance requirements.

1.3.4 Customization and Control

Managed APIs expose a fixed set of models with predetermined parameters. You get what the provider offers. When you control the stack, you choose everything:

- **Model selection**: Run any open-weight model—Llama, Mistral, Qwen, Phi, or specialized models for code, math, or specific languages. Switch models without changing providers or renegotiating contracts.
- **Fine-tuning**: Adapt models to your domain with your data. A legal firm can fine-tune on case law; a medical organization can specialize for clinical terminology. The fine-tuned weights stay on your infrastructure.
- **Inference parameters**: Control temperature, top-p, repetition penalties, and stopping conditions precisely. Some parameters available on your own deployment aren't exposed by managed APIs.
- **No quotas**: Managed APIs impose rate limits and usage caps, especially on newer or more capable models. Your infrastructure has no artificial limits—only the throughput your hardware can sustain.
- **Version pinning**: Lock to a specific model version for reproducibility. Managed APIs occasionally update models, subtly changing behavior. With your own deployment, the model only changes when you change it.

This control matters most when AI is core to your product rather than a peripheral feature. If model behavior directly affects your business outcomes, controlling that behavior becomes a strategic priority.

1.3.5 When NOT to Self-Host

Self-hosting isn't always the right choice. Managed APIs win in several scenarios:

- **Low or unpredictable volume**: If you're making hundreds of requests per day rather than thousands, the operational overhead of maintaining inference infrastructure exceeds the cost savings. APIs let you pay only for what you use.
- **Experimentation phase**: When you're still figuring out whether AI adds value to your product, managed APIs let you iterate without infrastructure commitment. Build the prototype first; optimize deployment later.
- **Cutting-edge models**: The best open-weight models lag behind frontier closed models by 6–12 months. If you need GPT-5 or Claude Opus capabilities today, APIs are your only option.

- **Multimodal requirements**: Open models for vision, audio, and video generation are maturing but still trail proprietary alternatives in quality. Complex multimodal pipelines may require API access.
- **Limited engineering capacity**: Running inference infrastructure requires systems engineering skills—debugging GPU memory issues, optimizing throughput, maintaining uptime. If your team lacks this expertise and can't acquire it, the learning curve may not justify the benefits.

The decision isn't binary. Many organizations use managed APIs for experimentation and frontier capabilities while self-hosting for high-volume production workloads with stable requirements. Chapter **??** includes a decision framework for evaluating which workloads to self-host.

1.4 What You'll Build in This Book

This book teaches inference systems by building one. You'll construct a control plane—the software layer that manages model serving—and evolve it from a simple proxy to a production-grade platform. The models grow alongside the infrastructure: from 7B parameters on a single GPU to 400B distributed across a cluster.

1.4.1 The Progressive Journey

The book is organized into four parts, each targeting a larger model size and adding capabilities to the control plane:

Part I: Foundations (7B) You'll run your first model, understand hardware requirements, learn model formats and quantization, compare inference engines, and build Control Plane v0.1—a basic proxy with health checks and metrics. A single consumer GPU or cloud instance handles everything.

Part II: Production (30B) The control plane gains auth, rate limiting, caching, and request queuing. You'll deploy 30B models requiring careful memory management and learn optimization techniques that squeeze maximum throughput from your hardware.

Part III: Multi-Tenant (70B) Control Plane v0.3 supports multiple tenants with isolated quotas, usage tracking, and billing integration. You'll distribute 70B models across multiple GPUs using tensor parallelism and build routing logic that directs requests to appropriate backends.

Part IV: Inference Lab (400B) The capstone: a 400B parameter deployment on H100 GPUs, serving a coding assistant called CodeLab. Control Plane v1.0 includes everything needed to operate inference at scale—the complete system you've built across the book.

1.4.2 Control Plane Evolution

The control plane is written in Go. This choice is deliberate: Go compiles to a single binary with no runtime dependencies, making deployment straightforward. Its concurrency model handles many simultaneous inference requests efficiently. And Go is readable—you don't need deep language expertise to understand or modify the code.

The architecture uses interfaces to abstract components. An `InferenceBackend` interface defines how the control plane talks to inference engines; implementations exist for Ollama, llama.cpp, and vLLM. When you add vLLM support in Part II, you implement the interface without changing the rest of the system. This pattern repeats throughout: `RateLimiter`, `Cache`, `Router`, `BillingProvider`—each is an interface with swappable implementations.

Table 1.2 Control plane evolution across the book.

Version	Part	Key Capabilities
v0.1	I	Single backend, health checks, Prometheus metrics
v0.2	II	Auth, rate limiting, caching, request queue
v0.3	III	Multi-tenant, usage tracking, billing, model routing
v1.0	IV	Distributed inference, advanced scheduling, complete platform

Each version builds on the previous. Code from Chapter 5 remains in the final system; it just gains neighbors. This mirrors how production systems actually evolve—incrementally, with backward compatibility.

1.4.3 What You'll Learn Beyond the Code

The control plane is the tangible artifact, but the deeper goal is building intuition. You'll learn enough about transformer memory layout to calculate VRAM requirements for any model. You'll understand quantization well enough to choose between Q4_K_M and Q5_K_S for your workload. You'll grasp attention mechanisms sufficiently to reason about KV cache growth and context length trade-offs.

This isn't a machine learning theory book—we won't derive backpropagation or prove convergence bounds. But we cover the systems-relevant math: memory calculations, throughput modeling, parallelism strategies. When new techniques emerge—mixture of experts, speculative decoding, expert parallelism—you'll have the mental framework to evaluate them. Chapter **??** implements expert parallelism from scratch, not just to use it, but to understand when and why it helps.

The field moves fast. Models that seem large today will be routine tomorrow. The specific numbers in this book will date; the systems thinking won't. Our aim is engineers who can read a new model's architecture paper and estimate what hardware it needs, or evaluate a new inference engine and predict where it will excel.

1.5 Hands-On: Your First Inference

Enough theory—let's run a model. We'll use Ollama, an inference server that handles model downloads, format conversion, and serving behind a simple API. It's not the fastest option (we'll cover vLLM and llama.cpp later), but it's the easiest path from zero to working inference.

By the end of this section, you'll have a 7B parameter model running and responding to requests.

1.5.1 Installing Ollama

Ollama runs on macOS, Linux, and Windows. Installation takes under a minute:

macOS Install via Homebrew:

```
brew install ollama
```

Linux Use the install script:

```
curl -fsSL https://ollama.com/install.sh | sh
```

Windows Download the installer from `https://ollama.com/download/windows`.

After installation, start the Ollama server:

```
ollama serve
```

The server runs on `localhost:11434` by default. In a separate terminal, verify it's running:

```
curl http://localhost:11434/api/tags
```

You should see an empty model list: `{"models":[]}`. We'll fix that next.

1.5.2 Running Your First Model

Ollama downloads models on demand from its registry. We'll use Qwen 2.5, Alibaba's open model family that offers strong performance across reasoning, code, and multilingual tasks [11]. The 7B variant balances capability with modest hardware requirements:

```
ollama pull qwen2.5:7b
```

The download is approximately 4.7GB (the model is quantized to 4-bit precision by default). Once complete, you can interact with it directly:

```
ollama run qwen2.5:7b
```

This opens an interactive chat session. Type a message, press Enter, and watch tokens stream back. Behind the scenes, Ollama loads the model weights into memory (GPU if available, CPU otherwise), processes your input through the transformer, and generates tokens autoregressively—exactly the process we described in Section 1.2.

Type `/bye` to exit the session.

To see what models you have locally:

```
ollama list
```

The output shows model name, size, and when it was last modified. As you work through this book, you'll accumulate models here—different sizes, different quantizations, different families. We use Qwen throughout the book for consistency; Appendix **??** provides equivalent commands for Mistral models if you prefer a European alternative.

1.5.3 Making API Requests

The interactive CLI is useful for experimentation, but applications call Ollama's HTTP API. The server exposes REST endpoints for generation, chat, and model management.

A basic generation request:

```
curl http://localhost:11434/api/generate \
  -d '{
    "model": "qwen2.5:7b",
    "prompt": "Explain what AI inference is in one paragraph.",
    "stream": false
  }'
```

The response is JSON containing the generated text and timing metadata:

```
{
  "model": "qwen2.5:7b",
  "response": "AI inference refers to the process of ....",
  "done": true,
  "context": [ 151644, 8948, 198, ... ],
  "total_duration": 1930969541,
  "load_duration": 41119208,
  "prompt_eval_count": 39,
  "prompt_eval_duration": 112548500,
  "eval_count": 80,
  "eval_duration": 1776717000
}
```

Setting `stream: false` waits for the complete response. For interactive applications, streaming provides better perceived latency—users see tokens as they're generated:

```
curl http://localhost:11434/api/generate \
  -d '{"model": "qwen2.5:7b", "prompt": "Write a haiku.", "stream": true}'
```

This returns newline-delimited JSON objects, one per generated token.

For chat-style interactions with conversation history, use the `/api/chat` endpoint:

```
curl http://localhost:11434/api/chat \
  -d '{
    "model": "qwen2.5:7b",
    "messages": [
      {"role": "user", "content": "What is the capital of France?"}
    ],
    "stream": false
  }'
```

The API accepts additional parameters—`temperature`, `top_p`, `num_predict`—covered in Chapter **??**. The defaults work well for initial experimentation.

1.6 Understanding the Response

The response from your first inference contains more than just text. The metadata reveals how the model processed your request—information that becomes essential for optimization and debugging.

1.6.1 Tokens and Tokenization

Language models don't process text character by character. They operate on *tokens*—subword units that balance vocabulary size against sequence length. The word "understanding" might be a single token; "tokenization" might split into "token" + "ization." Common words compress to single tokens; rare words decompose into pieces.

Different models use different tokenizers. Qwen 2.5 uses a byte-pair encoding (BPE) tokenizer with approximately 150,000 tokens in its vocabulary. This affects everything from input costs (APIs charge per token) to context limits (models have token budgets, not character budgets).

In the response above, `prompt_eval_count: 39` indicates your prompt consumed 39 tokens. The phrase "Explain what AI inference is in one paragraph." contains 8 words but 39 tokens—the tokenizer made different choices than word boundaries. The field `eval_count: 80` shows the model generated 80 tokens in response.

You can inspect tokenization directly:

```
curl http://localhost:11434/api/generate \
  -d '{"model": "qwen2.5:7b", "prompt": "Hello world", "raw": true}'
```

Understanding tokenization matters for three reasons. First, cost: API pricing is per-token, so knowing how your prompts tokenize helps budget. Second, context: a 4096-token context window accommodates different amounts of text depending on content. Code typically tokenizes more efficiently than prose; non-English text often tokenizes less efficiently. Third, performance: longer token sequences mean more computation. A 1000-token prompt requires more processing than a 100-token prompt, affecting latency.

1.6.2 Timing Metrics

The response includes timing data in nanoseconds:

- `total_duration`: Wall-clock time for the entire request—1.93 seconds in our example.
- `eval_duration`: Time spent generating tokens—1.78 seconds. The difference (150ms) covers prompt processing and overhead.

From these numbers, you can calculate *tokens per second*: 80 tokens/1.78 seconds ≈ 45 tok/s. This metric—generation throughput—is the primary performance indicator for inference systems. Higher is better; it directly affects how fast users see responses.

Two latency metrics matter for interactive applications:

Time to First Token (TTFT) How long until the first generated token appears. For streaming responses, this is perceived latency—users see something immediately even if generation continues. TTFT includes prompt processing time.

Inter-Token Latency (ITL) Average time between generated tokens. With 80 tokens in 1.78 seconds, ITL is approximately 15ms. Lower ITL means smoother streaming.

Hardware dramatically affects these numbers. The same Qwen 2.5 7B model might achieve 20 tok/s on CPU, 75 tok/s on an RTX 4060, and 150 tok/s on an RTX 4090. Chapter **??** provides benchmarks across GPU tiers.

1.6.3 Generation Parameters

Beyond the prompt, several parameters control how the model generates text. These aren't visible in our simple request because Ollama used defaults, but they're available when you need them:

- **temperature**: Controls randomness. At 0, the model always picks the highest-probability token (deterministic). At 1.0, sampling follows the probability distribution. Higher values (up to 2.0) increase creativity and unpredictability. Default is typically 0.7–0.8.
- **top_p** (nucleus sampling): Restricts sampling to tokens whose cumulative probability exceeds this threshold. At 0.9, the model considers only the most likely tokens that together account for 90% of probability mass. Lower values increase focus; higher values allow more variety.
- **num_predict**: Maximum tokens to generate. Limits response length. The model may stop earlier if it generates a stop token.

For most applications, defaults work well. Adjust temperature for creative tasks (higher) or factual extraction (lower). Chapter **??** explores these parameters in depth with guidance on tuning for specific use cases.

1.7 Summary

This chapter established the foundation for everything that follows. You now understand the fundamental distinction between training and inference: training creates models through expensive, iterative optimization across massive datasets; inference uses those models for prediction through forward passes alone. This asymmetry—training costs millions, inference runs on a laptop—is why self-hosted inference is practical.

You've evaluated the trade-offs. Self-hosting offers predictable costs, data control, lower latency, and freedom to customize. It requires systems knowledge and operational responsibility. Managed APIs remain appropriate for low-volume workloads, cutting-edge models, and teams without infrastructure expertise. The choice depends on your requirements—and often, organizations use both.

Most importantly, you've run your first model. Qwen 2.5 7B is now responding to requests on your machine. You understand the response structure: tokens as the atomic unit, timing metrics that reveal performance, and parameters that control generation behavior.

Key Takeaways

- Inference runs only forward passes—no gradients, no training data, achievable on consumer hardware.
- Self-hosted inference means controlling your stack, whether that's cloud instances, colocated servers, or local workstations.
- Open models (Qwen, Mistral, Llama, Phi, Gemma) make self-hosting viable for many production workloads.
- Tokens are the fundamental unit: they affect cost, context limits, and performance.
- Key metrics: tokens per second (throughput), time to first token (responsiveness), inter-token latency (streaming smoothness).

Next: How LLMs Work

Before diving into hardware requirements, Chapter **??** explains what's actually happening inside these models. You'll understand the transformer architecture from a systems perspective—not the math, but the components and data flow that determine memory usage and performance. We'll trace the inference lifecycle from prompt to response, and most importantly, you'll learn why KV cache exists and how it affects everything from memory requirements to context length limits.

Then Chapter **??** answers the question you're likely asking: what hardware do I actually need? We'll calculate memory requirements for different model sizes, compare GPU tiers and their price-performance ratios, and help you choose infrastructure that matches your workload.

Problems

1.1 Cost Comparison

Calculate the monthly cost of running 100,000 inference requests through OpenAI's GPT-4 API versus self-hosting a 7B model on an RTX 4060. Assume average request length of 500 tokens input and 200 tokens output.

1.2 Latency Measurement

Using Ollama, measure the time-to-first-token (TTFT) and tokens-per-second for three different prompts: (a) a simple question, (b) a code generation request, and (c) a creative writing prompt. What patterns do you observe?

1.3 Streaming vs Non-Streaming

Implement a simple Python script that makes both streaming and non-streaming requests to Ollama. Measure the perceived latency (time until user sees first output) for each approach.

References

1. Touvron, H., Martin, L., Stone, K., et al.: Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv preprint arXiv:2307.09288 (2023). https://arxiv.org/abs/2307.09288
2. Jiang, A.Q., Sablayrolles, A., Mensch, A., et al.: Mistral 7B. arXiv preprint arXiv:2310.06825 (2023). https://arxiv.org/abs/2310.06825
3. Bai, J., Bai, S., Chu, Y., et al.: Qwen Technical Report. arXiv preprint arXiv:2309.16609 (2023). https://arxiv.org/abs/2309.16609
4. Li, Y., Bubeck, S., Eldan, R., et al.: Textbooks Are All You Need II: phi-1.5 Technical Report. arXiv preprint arXiv:2309.05463 (2023). https://arxiv.org/abs/2309.05463
5. Gemma Team: Gemma: Open Models Based on Gemini Research and Technology. arXiv preprint arXiv:2403.08295 (2024). https://arxiv.org/abs/2403.08295
6. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016). https://www.deeplearningbook.org/
7. Vaswani, A., Shazeer, N., Parmar, N., et al.: Attention Is All You Need. Advances in Neural Information Processing Systems 30 (2017). https://arxiv.org/abs/1706.03762
8. OpenAI: API Pricing. https://openai.com/api/pricing/ (accessed December 2025)
9. Anthropic: Claude API Pricing. https://www.anthropic.com/pricing (accessed December 2025)
10. OpenAI: API Data Usage Policies. https://openai.com/policies/api-data-usage-policies (accessed December 2025)
11. Qwen Team: Qwen2.5: A Party of Foundation Models. https://qwenlm.github.io/blog/qwen2.5/ (2024)