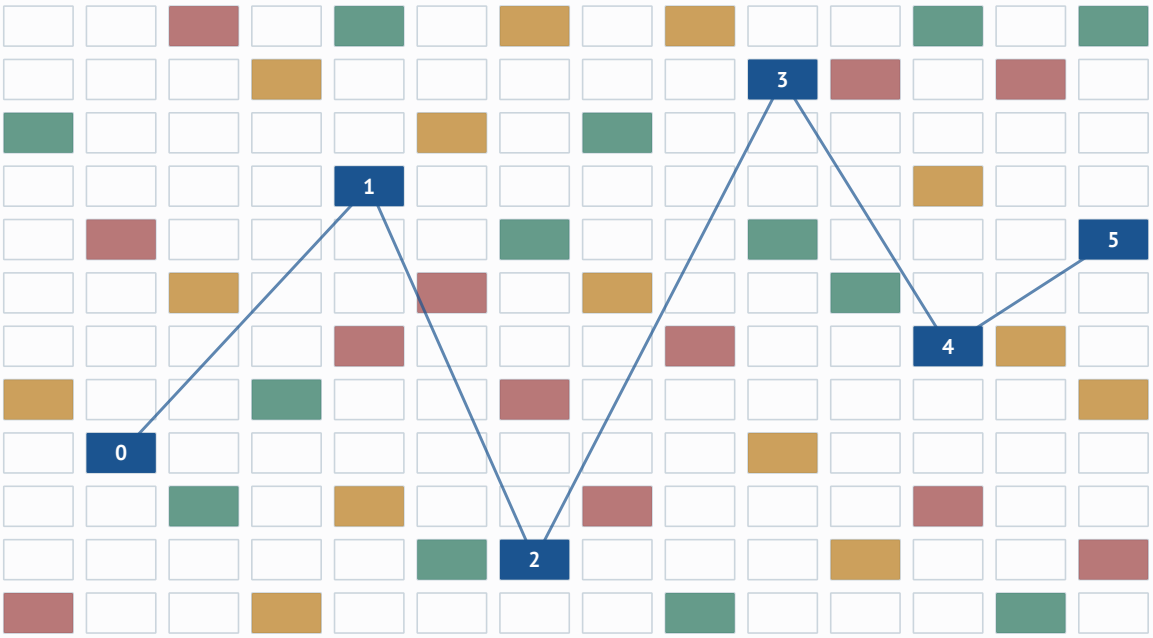
LAB GUIDE

# Enterprise AI Platform

GPU infrastructure, model serving and the operation of an AI platform inside a company

KV CACHE · ONE SEQUENCE, SIX BLOCKS




Thomas Zachmann

17 chapters · 4 appendices · 386 pages

# Enterprise AI Platform

Lab Guide — GPU Infrastructure, Model Serving and Operations

Thomas Zachmann
As of September 2026

**Enterprise AI Platform — Lab Guide**
GPU infrastructure, model serving and the operation of an AI platform inside a company

Thomas Zachmann
As of September 2026

# Contents

# About this book

## 0.1 Who this book is written for

This book is addressed to everyone who builds and operates a company-wide AI platform — and who brings production experience with Kubernetes, GitOps, identity providers, secret management and monitoring to the task.

That task is presenting itself in a great many organizations at once just now, and it mostly presents itself unplanned. Developers have long been using ChatGPT and Claude, often through private accounts. Business units have built proofs of concept that nobody operates. The data protection officer has questions. Procurement has seen an invoice that nobody can attribute to a team. And somewhere there is a server with two GPUs for which no operational process exists.

Whoever is to turn that situation into an operable platform needs different knowledge from the data scientist who selects models, and different knowledge from the application developer who programs against an API. What is needed is knowledge about GPUs in the cluster, about the mechanics of inference servers, about access control on models, about cost attribution — and about the question of which data may reach which model. This book gathers exactly that knowledge.

## 0.2 What this book is not

That is important enough for a section of its own, because it determines what stands on the following pages — and above all what does not.

**This is not a Kubernetes book.** It does not explain what a pod, a deployment, a service or a namespace is. It does not explain how the scheduler works in principle, how ingress works, or how to write a Helm chart. Where this book talks about Kubernetes, it is exclusively about what is different for GPU and inference workloads than it is for ordinary applications — and that is a great deal.

**This is not a DevOps primer.** CI/CD, GitOps, infrastructure as code, containers, registries, network segmentation and observability are taken as known. They appear, but always in their AI-specific form: ArgoCD not explained as a tool, but in the question of how to move a 16-gigabyte model artefact sensibly through a GitOps pipeline.

**This is not a machine learning book.** You will not learn here how a transformer works, how attention is defined mathematically, or how to train or fine-tune a model. The inside of the model is of interest only so far as it determines operational decisions — and that is surprisingly far: without an understanding of why the KV cache grows linearly with the context length, no GPU capacity can be planned. But the boundary runs clearly along the question: does this knowledge influence a decision that a platform operator takes?

**This is not a tutorial.** A tutorial leads you from beginning to end once and is used up afterwards. This book is laid out as a reference. Most chapters are built so that one

can jump into them directly: concept, reference, lab, operations and troubleshooting, decision matrix, interview questions. Chapters 1 and 17 place things in context and contain no lab; Chapter 9 treats a topic that cannot be carried out in the reference lab and therefore appears as a concept with a configuration reference. You may well read it linearly the first time. It becomes useful the twelfth time, when at 11 at night you look up why a pod with `nvidia.com/gpu: 1` is hanging in `Pending`.

## 0.3 What is assumed

| Area | Level expected |
|---|---|
| Linux | Confident on the command line, systemd, kernel modules, package management |
| Containers | Building images, registries, runtimes, volumes, network modes |
| Kubernetes | Production experience: workloads, scheduling, RBAC, storage, network |
| GitOps / CI-CD | ArgoCD or Flux in use, pipeline design |
| Identity | OIDC fundamentals, an IdP such as Keycloak in production |
| Observability | Prometheus, Grafana, logs and traces in daily use |
| Network and security | TLS, NetworkPolicies, egress control, secret handling |
| Python | Reading and adapting — not developing |
| Machine learning | Nothing. It is built up where it is needed. |

Anyone who is unsure about several of these rows should close those gaps first. This book builds on them and does not repeat them.

## 0.4 The reference lab

All the practical exercises in this book refer to one concrete, deliberately small environment. That is a decision against arbitrariness: instructions meant to work on every conceivable platform end up working properly on none, because they leave out precisely the points of friction on which one actually fails.

| Component | Form in the reference lab |
|---|---|
| Virtualization | Proxmox VE, several virtual machines on one host |
| GPU | One NVIDIA GPU of the RTX class with 24 GiB of VRAM, passed through to a VM via VFIO |
| Kubernetes | RKE2, several nodes, the GPU on exactly one worker |
| Models | Llama 3.1 8B and Qwen 2.5 from 0.5B to 32B, various quantizations |

This environment is not a makeshift but is, in one respect, even more realistic than a cloud cluster with eight H100s: it forces decisions. 24 GiB of VRAM fills up quickly. One GPU cannot be shared out among teams at will. That scarcity is exactly what platform work consists of — and whoever masters it in the small can justify it in the large.

Where the hardware reaches its limit, this book says so explicitly. Tensor parallelism across several GPUs, for instance, cannot be carried out with one card. Such topics are then treated as a concept with a full configuration reference and expressly marked as not executable in the lab. What you will not find here is a lab that works in the book and not on your hardware.

## 0.5 Conventions

Recurring elements structure every chapter. You recognize them by their colour and their label.

**LAB** **This is what a lab looks like**

*Goal: What is meant to work at the end*

Practical exercises in the reference lab. Always with the expected output, so that you can tell whether it worked — and how you notice that it did not.

**KEY POINT**

Core statements that carry the rest of a section. If you only skim a chapter, read these boxes.

**CAUTION**

Pitfalls with real consequences: data loss, an outage, a cost explosion or a security hole. These boxes almost always arise from mistakes somebody has already made.

**INTERVIEW QUESTION**

*And this is what an interview question looks like.*

At the end of every chapter there are questions at senior level, each with what makes a weak, a good and a genuinely convincing answer.

Every chapter also carries a version stamp at the top. It names the software versions against which it was written and checked:

**WRITTEN AGAINST**

Kubernetes 1.31 · RKE2 v1.31.x · vLLM 0.8.x

That is not decoration. The toolbox described here changes fast — vLLM publishes releases with new engine arguments regularly, KServe moves API fields, the GPU Operator changes defaults. A reference work without version information ages unnoticed and becomes dangerous through that. With version information it ages visibly and stays usable, because you know what you have to check against.

## 0.6 Reading paths

There are three sensible ways through this book.

**The build path** you are to build a platform. Read linearly. The order of the chapters matches the order in which the layers build on one another: first understand what inference costs, then make GPUs available, then serve models, then access and governance, and operations last.

**The interview path** you are preparing for conversations. Read the opening *Goal* box and the *Key point* boxes of every chapter, then Appendix A as a whole, and from there go back into the chapters whose questions you could not answer confidently.

**The reference path** you have a concrete problem. Use the table of contents and the index, jump into the troubleshooting part of the chapter in question — each of Chapters 2 to 16 has one; the contextual Chapters 1 and 17 do not. Where a separate reference section exists as well, it is named as such in the table of contents. Those sections are written so that they work without the rest of the chapter.

## 0.7 A note on shelf life

A book about a technology that moves on a monthly rhythm has an obvious problem. This book deals with it by separating two levels clearly.

The one level is concepts: why prefill and decode are fundamentally different workloads. Why the KV cache is the actual capacity problem. Why an AI gateway is the right place for governance. Why GPU utilization is the central FinOps metric. That level is stable. It was right three years ago and will be right in three years.

The other level is concrete flags, field names and version states. That level ages, and it ages fast. It is therefore consistently moved out into reference sections and version stamps instead of being scattered through the running text. When a command no longer holds, you find it at a clearly marked place — and the thought around it carries on.

Whoever learns only commands has to throw this book away in a year. Whoever understands the concepts can look the commands up in the current documentation and place them correctly. The second is the aim.

# About the author

In technical books a page about the writer usually stands at this point. It is dispensed with here. Whoever writes a book about platform work should be measured by the work and not by his description of himself.

Anyone who would nevertheless like to know who stands behind these chapters will find it in the following places — and more dependably than a biography could be.

The author works freelance and supports organizations in building and operating the platforms described here — from the architecture through the implementation to the handover into operations, likewise in reviewing existing installations and in training on the material of these chapters. Enquiries about that are welcome, as are questions about the book, corrections and reports of mistakes: a reference work gets better through contradiction, not through agreement.


**CONTACT AND WORK**

| | |
|---|---|
| **Web** | thomaszachmann.de |
| **Company** | nyrvex.com |
| **Code** | gitlab.com/thomaszachmann |
| **E-mail** | thomas@zachmann.work |

PART I

# LLM Inference Fundamentals for Infrastructure Engineers

*Before a GPU goes into the cluster, it has to be clear what actually happens on it. This part explains LLM inference from the perspective of whoever has to operate it — not from the perspective of whoever trains models.*

CHAPTER 1

# The Enterprise AI Platform in Overview

| | |
|---|---|
| **GOAL** | State the role of the Enterprise AI Platform Engineer precisely: responsibility, demarcation from neighbouring roles, and the question of which part of existing platform knowledge carries over and which does not. |
| **ASSUMES** | Production experience with Kubernetes and platform engineering. No prior knowledge of GPUs, models or inference. |
| **YOU WILL BE ABLE TO** | Explain the full stack map of this book, name the five structural differences from classical platform work, and judge which chapters are relevant to a concrete undertaking. |

**WRITTEN AGAINST**

As of September 2026

This chapter contains no configuration and no lab. It builds the map that every following chapter refers to. Anyone impatient enough to want GPUs in a cluster immediately can start at Chapter 4 and come back here as soon as the question arises of why the platform is cut the way it is.

## 1.1 What this role is responsible for

The role of the Enterprise AI Platform Engineer covers building and operating the infrastructure on which AI models are used within an organization. It is not responsible for which model answers well, but for models being available at all, for their use being controlled, for that use staying affordable, and for it demonstrably following the company's rules.

That can be sharpened into four questions. Anyone operating an AI platform has to be able to answer them at any time:

1. **Which model runs on which hardware — and who shares that hardware?** This is a capacity and isolation question, and it is considerably harder for GPUs than for CPU and RAM.
2. **How does a model get into production reproducibly, and out again?** This is a delivery question, but with artifacts of double-digit gigabyte size and startup times in minutes rather than seconds.
3. **Who may use which model with which data?** This is an identity, authorization and governance question — and the point at which an AI platform differs most clearly from an ordinary platform.
4. **What does operation cost, and to whom is it charged?** This is a FinOps question which, unlike classical workloads, arises not monthly but per request and is measurable per request.

**KEY POINT**

The role is not an extension of the data science role downwards but an extension of the platform engineering role upwards. The professional core remains infrastructure: provide resources, control access, secure operation, allocate cost. What is new is the kind of resource — and the fact that the content of a request is suddenly a compliance object.

## 1.2 Demarcation from neighbouring roles

The term "AI engineer" is currently used for very different activities. For positioning yourself — towards employers as towards clients — it pays to draw the distinctions sharply.

| Role | Responsible for | Typical question |
|---|---|---|
| Data scientist | Model selection, evaluation, data quality, quality of results in the domain | Is the answer right enough for the use case? |
| ML engineer | Training, fine-tuning, feature pipelines, experiment tracking | How do data become a better model? |
| AI engineer / LLM app developer | Applications against model APIs, prompting, agents, RAG logic | How do I build a product on top of a model? |
| MLOps engineer | The path from training to deployment, model registry, reproducibility | How does a trained model reliably reach operation? |
| **AI platform engineer** | **Runtime platform: GPUs, serving, gateway, identity, governance, observability, cost** | **How do I operate models for the whole organization — securely, reliably and affordably?** |

The boundary to the MLOps engineer is blurred in practice and in smaller organizations often the same person. The difference in emphasis is real nonetheless: MLOps thinks from the model artifact, platform engineering from the runtime environment and its users. This book takes the second perspective throughout.

**KEY POINT**

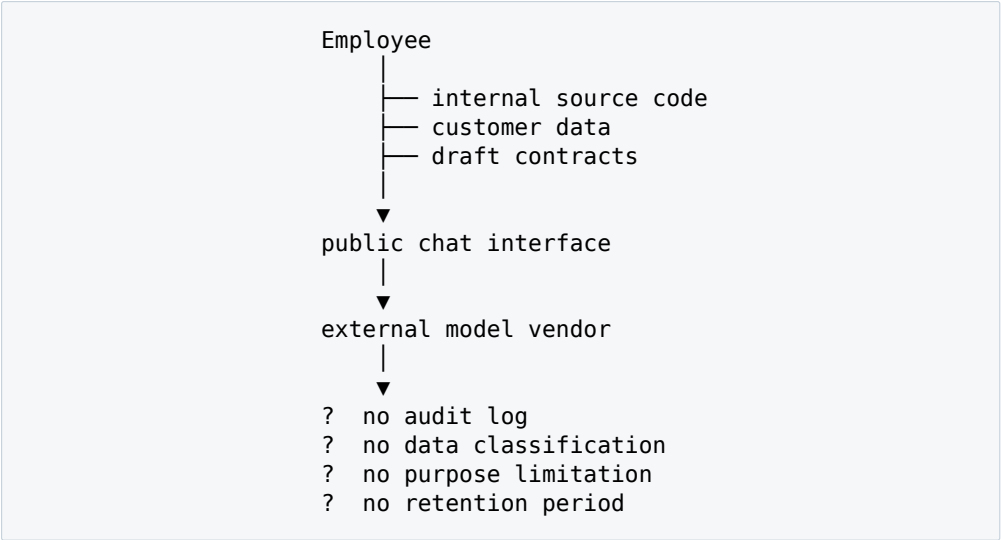
For positioning yourself, the decisive statement is not "I can do AI too" but: **I build and operate the platform on which AI runs in this company.** That is an infrastructure statement, and it is not weaker than a data scientist's but complementary to it.

## 1.3 Why this role is emerging now

Roles do not emerge from technology alone but from pressure. For AI infrastructure this pressure comes down to four drivers, which act at the same time in almost every larger organization.

### 1.3.1 Loss of control through shadow AI

The most widespread starting point is not a platform but the absence of one. Employees use public chat interfaces with private or team-owned accounts. What happens then is, from an information security point of view, uncomfortably simple:




Shadow AI: the uncontrolled path a platform has to replace

The problem is not the model vendor — many vendors meet high standards on their enterprise tariff. The problem is that the organization knows nothing about the process. There is no evidence of which data left the building, no way to withdraw access, and no basis for answering a customer or a supervisory authority.

A platform does not solve this by forbidding. It solves it by making the controlled path more convenient than the uncontrolled one. That is an infrastructure task.

### 1.3.2 Cost that belongs to nobody

AI usage produces cost per request. That is a structural difference from almost every other workload on a platform. A Kubernetes cluster costs a sum per month, largely independent of how intensively it is used. Model access costs in proportion to usage — and in an order of magnitude that comes as a surprise as soon as several hundred developers use it seriously.

Without a platform this bill lands in procurement as a single sum, with no allocation to team, application or purpose. It is therefore neither steerable nor negotiable. In practice the wish for cost allocation is frequently what gets an AI gateway project approved at all — more often than security arguments.

### 1.3.3 Data sovereignty and regulation

For part of the data the question is not whether the vendor is trustworthy but whether the data may leave the building at all. Health data, personnel matters, ongoing legal proceedings, security-relevant design data, source code with competitive relevance: for these categories there are requirements that no contract with an external vendor dissolves.

From this an architecture follows almost inevitably that is hybrid — external models for uncritical tasks, locally operated models for the rest — and with it the necessity of running your own GPUs. It is exactly at this point that a procurement question becomes a platform question.

### 1.3.4 Dependence on individual vendors

The fourth driver is operational in nature. When a growing number of internal applications are programmed directly against the API of a single vendor, the organization has an availability problem and a negotiating problem at the same time. One outage hits everything at once; a price or model change forces changes in every application.

The answer is an abstraction layer between application and vendor. That is not a new pattern — it is the same argument with which one introduces a reverse proxy, a message broker or a service mesh. Only here it is called an AI gateway.

## 1.4 The four maturity levels of an AI platform

Organizations do not reach a platform in one step. In practice four levels can be distinguished, and it pays to place your own organization honestly — because each level suggests a different next measure.

This classification is also a good conversational instrument with clients: it turns a diffuse enquiry ("we want to do something with AI") into a concrete starting position.

**Level 0 — shadow AI** There is no platform. Usage takes place, but invisibly, through private or team-owned accounts. There is no inventory, no audit log, no cost allocation. What is characteristic is that nobody can answer how many people in the building actually use AI.

**Level 1 — central access** An AI gateway bundles access. Identity, quotas and logging exist, but the models run entirely at external vendors. This is the point at which visibility arises — and with it, for the first time, dependable figures for every further decision.

**Level 2 — hybrid** In addition, your own models run on your own GPUs. The gateway decides which request goes where. From here on the organization needs the full GPU and serving stack — and therefore the role this book is about.

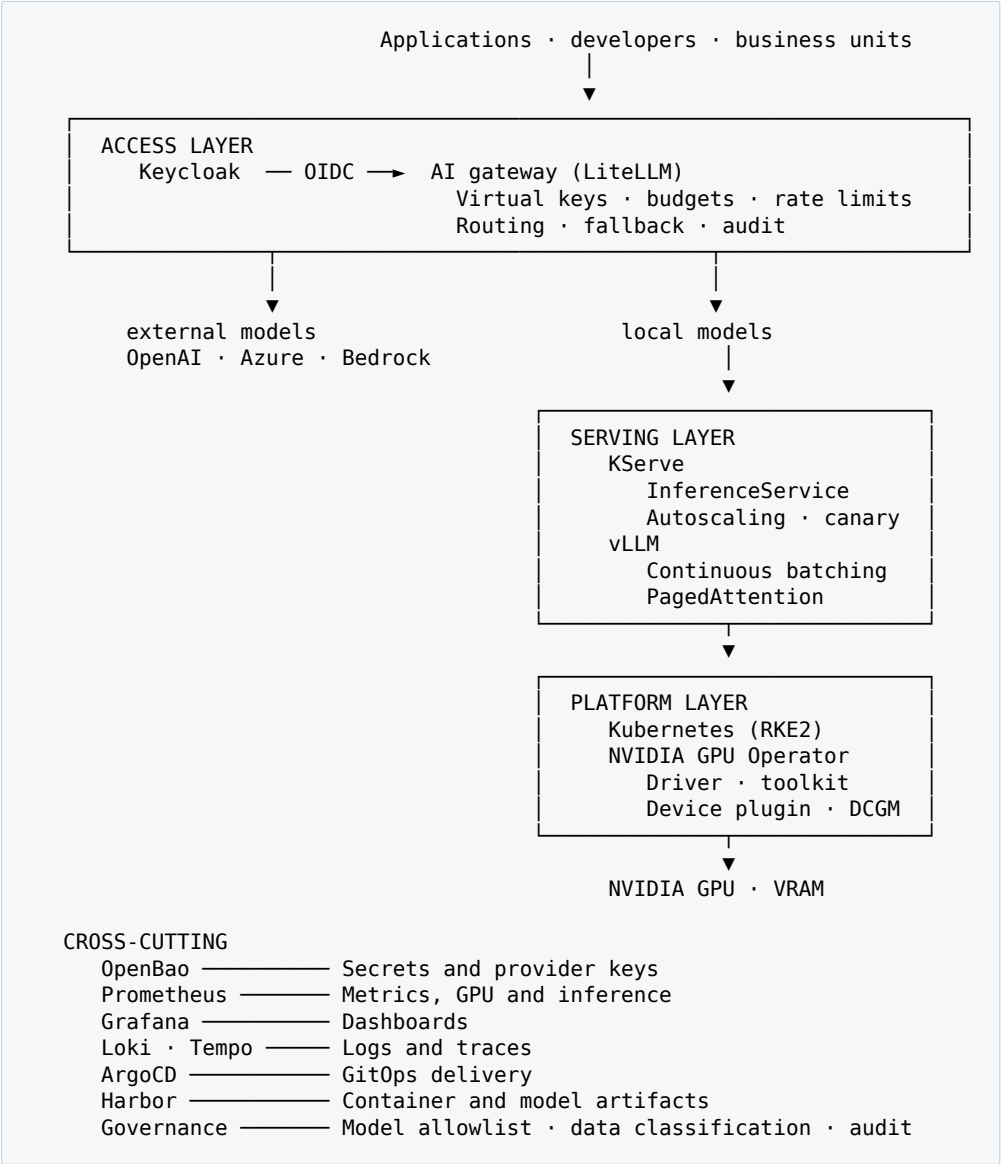
**Level 3 — platform with governance** Several teams use the platform as a self-service offering. There is tenant isolation, model allowlists by data classification, chargeback, SLOs and a defined path by which a new model goes into operation. That is the target state this book describes.

| Level | Visibility | Typical effort | Central risk |
|---|---|---|---|
| 0 — shadow AI | none | – | Data outflow, nothing can be evidenced |
| 1 — gateway | complete for external use | weeks | Gateway becomes a single point of failure |
| 2 — hybrid | own inference in addition | months | Poor GPU utilization makes it expensive |
| 3 — governance | including data classification and cost per team | ongoing | Complexity overwhelms the operations team |

> **KEY POINT**
>
> The order is not arbitrary. Anyone procuring GPUs immediately at level 0 buys hardware without usage data and stands a good chance of sizing it wrongly. The path through level 1 delivers exactly the figures with which the purchase can subsequently be justified and correctly dimensioned.

## 1.5 The stack map

Everything else in this book fits into the picture below. It is the target architecture that the chapters work towards step by step.



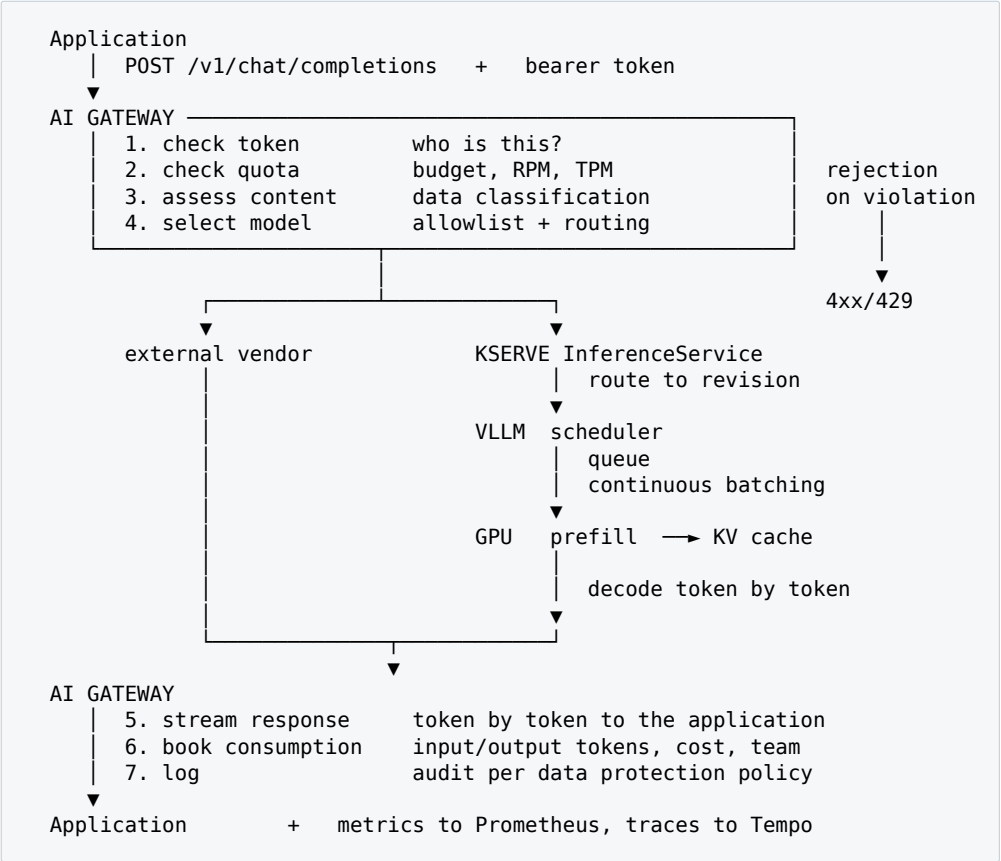

The target architecture of an Enterprise AI Platform

The decisive observation about this picture is how it is distributed: the greater part consists of building blocks an experienced platform engineer already operates. Kubernetes, identity, secrets, network, monitoring, registry, GitOps — none of these are AI topics. What is genuinely new is four boxes: the GPU stack, the inference server, the serving abstraction above it and the AI gateway in front of it.

> **KEY POINT**
>
> Roughly two thirds of an Enterprise AI Platform is classical platform engineering. That is the real good news: the new part is narrow and deep, not broad and shallow — and it comes down to four building blocks and five structural differences.

## 1.6 A request on its way through the platform

The layer diagram shows what exists. It does not show what happens. The walk-through below follows a single request from the application to the GPU and back — naming, at every step, which chapter covers it. Anyone who can tell this sequence freely has understood the architecture.




The path of a request, with the decisions at every station

| No. | Station | What is decided here | Chapter |
|---|---|---|---|
| 1 | Authentication at the gateway | Is the token valid? Does it belong to a person or a machine? From which team? | 12 |
| 2 | Quota check | Is the team's budget exhausted? Are requests per minute or tokens per minute exceeded? | 11 |
| 3 | Content assessment | Does the prompt contain personal data or content classified as confidential? | 13 |
| 4 | Model selection | Which models may this user employ with this data class? External or local? What is the fallback on failure? | 11, 13 |
| 5a | KServe | Which model version is routed to? Does a running instance exist or does one have to start first? | 10 |
| 5b | vLLM scheduler | Is the request processed immediately or queued? Does it fit into the | 7 |

| No. | Station | What is decided here | Chapter |
|---|---|---|---|
| | | current batch? Is there enough free KV cache? | |
| 5c | GPU | Prefill processes the whole prompt at once, decode then produces token by token. | 2 |
| 6 | Consumption accounting | How many tokens were consumed, what does that cost, and which team is it charged to? | 11, 14 |
| 7 | Logging | What is stored — and what expressly is not? | 13, 14 |

Three things can be read off this sequence that matter for the rest of the book.

**First**, the gateway makes most of the decisions although it is technically the simplest component. It is the only place where identity, content, cost and target model are known at the same time. That makes it the natural home for governance — and at the same time the component whose failure stops everything.

**Second**, the interesting failure cases occur not at the end but in the middle. A request can fail on quota, on data classification, on a vendor outage, on an exhausted KV cache or on an instance that is still loading. Each of these cases needs its own behaviour and its own error message. Chapter 11 covers which of them are signalled to the application and how.

**Third**, the most informative metrics arise at the edges: cost and usage data at the gateway, utilization data at the GPU. Only brought together do they yield a picture — cost per thousand tokens in self-hosted operation, for instance, which can only be formed when consumption figures from the gateway and utilization figures from DCGM are brought together. Chapter 14 shows how.

## 1.7 Familiar building blocks, changed properties

This narrative works well as an entry point into an architecture conversation: it is concrete enough to provoke follow-up questions and complete enough to accommodate every component.

The following table is the practical core of this chapter. It sets familiar platform components against their counterpart on an AI platform and names what is different about them.

| Classical | On an AI platform | The difference |
|---|---|---|
| CPU and memory requests | `nvidia.com/gpu` as an extended resource | Not divisible, not overcommittable, no limits in the usual sense. A GPU belongs to one pod — or is shared via MIG, time-slicing or MPS, with very different isolation guarantees. |
| Node pools and taints | GPU node pools | Conceptually identical. What is new is the variety of labels from GPU feature discovery and the need to distinguish GPU types in scheduling. |
| Container images | Model artifacts | Ten to over a hundred gigabytes, not in the image but loaded from object storage or Hugging Face. Startup time in minutes. Caching becomes a deployment topic. |

| Classical | On an AI platform | The difference |
|---|---|---|
| Horizontal pod autoscaler | KServe autoscaling | CPU utilization is largely worthless as a signal. What counts is queue depth, KV cache occupancy and time to first token. |
| Ingress and reverse proxy | AI gateway | Additionally token counting, budgets, model routing, provider fallback and cost allocation — at the level of request content, not only of transport. |
| OIDC and RBAC | Identity for model access | What is authorized is not only the endpoint but the combination of user, model and data classification. |
| Vault secrets | Provider API keys | Same mechanics, higher explosive force: a production key passed through is immediately convertible into money. |
| Prometheus metrics | DCGM and vLLM metrics | Same stack, different metrics. Utilization beats availability as the headline metric. |
| Blue-green and canary | Model rollouts | Same patterns, but expensive: two model versions in parallel mean twice the VRAM demand, not twice the RAM demand. |
| GitOps with ArgoCD | Model delivery | The manifest is in Git, the artifact is not. The reference to the model version becomes the critical, versioned link. |

## 1.8 The five fault lines

Where these analogies end, what is genuinely new begins. There are five structural differences, and they explain almost every decision taken in the chapters that follow.

### 1.8.1 GPUs are not fungible

Kubernetes treats CPU and memory as divisible, overcommittable quantities. A node with 16 cores can carry pods requesting 40 cores in total, as long as they do not all go flat out at once. None of that applies to GPUs. A GPU is allocated through the device plugin as an integer, exclusive resource. There is no “half a GPU” in the standard model, and there is no overcommitment the scheduler could absorb.

Covered in detail in Chapter 6, including the question of which method actually keeps which isolation promise.

Sharing is possible nonetheless — via MIG, time-slicing or MPS — but these methods differ fundamentally in what they guarantee. Time-slicing, for instance, shares compute time but **not** memory: two pods on the same card can drive each other into an out-of-memory state. Anyone who does not know this builds multi-tenancy that works in the load test and not in production. Chapter 6 covers this in detail.

### 1.8.2 Models are large, slow artifacts

A typical application container is a few hundred megabytes in size and starts in seconds. A model in `safetensors` format with eight billion parameters at 16-bit precision occupies around 16 gigabytes and has to be loaded into GPU memory completely before the first request can be answered.

Load time is the reason scale-to-zero is a considerably harder decision for LLMs than for ordinary services. See Chapter 10.

That changes operational assumptions which are otherwise taken for granted. Scale-to-zero is attractive because an idle GPU is expensive — but the cold start costs minutes, not milliseconds. Rolling updates briefly need capacity for two model versions. A node failure is no longer a matter of seconds. Chapters 8 and 10 cover the consequences.

### 1.8.3 Utilization is the headline metric, not availability

For classical services one optimizes for availability and accepts overcapacity in return. A web server at 15 percent CPU load is not a problem but a reserve.

For GPUs that is exactly a problem. A card utilized at 15 percent burns most of its purchase or rental price without return. GPU utilization is therefore not merely an operational metric but the central economic quantity of the whole platform — and the reason batching, sharing and scheduling take up so much room in this book.

> **CAUTION A common false conclusion**
>
> High GPU utilization in the sense of `nvidia-smi` does not automatically mean high throughput. The value displayed measures whether kernels are being executed at all — not how efficiently. An inference server with poor batching can show 100 percent and still deliver a fraction of the possible throughput. Chapter 14 covers which metrics are actually informative.

### 1.8.4 Cost arises per request

Classical platform cost is fixed cost at monthly granularity. AI cost is variable cost at request granularity — and, unlike almost everything else in operations, it can be attributed exactly to one person, one team and one application.

That is both an opportunity and an obligation. The opportunity: a platform can do genuine chargeback and thereby steer consumption. The obligation: as soon as this attribution is technically possible, it is expected. An AI gateway without cost allocation counts as incomplete in enterprise contexts. Chapters 11 and 14 cover both.

### 1.8.5 The request content is the compliance object

This is the deepest fault line. On an ordinary platform the content of a request is of no interest to the operator; it transports what applications send, and security means transport security and access control.

On an AI platform the content itself is the risk. Whether a prompt contains personal data, source code or contract details decides which model may process it. This creates requirements that classical platform work does not have in this form: content classification in the request path, model allowlisting by data category, egress control at application level, and the uncomfortable question of whether prompts may be logged at all — and for how long.

> **KEY POINT**
>
> Anyone who can explain the five fault lines has understood the conceptual core of this role. Everything else is tooling that can be exchanged. These five points remain, even when vLLM, KServe and LiteLLM go by different names in a few years.

## 1.9 What an AI platform costs

Hardly any question decides more often about a platform undertaking than this one — and hardly any is answered more often with numbers whose origin nobody can check. The method of calculation is therefore in the foreground here, not the result.

> **CAUTION On the numbers in this section**
>
> The values below are example assumptions illustrating the method, not current market prices. Vendor prices, hardware cost and electricity tariffs change continuously. Replace every line with your own values — the method stays valid, the numbers do not.

### 1.9.1 The demand calculation

The starting point is never the hardware but the usage volume. It can be derived from four quantities:

| Quantity | Assumption | Where it comes from |
|---|---|---|
| Users in total | 500 | Organization |
| of which active daily | 40 % | Experience, varies widely |
| Requests per active user per day | 30 | From gateway logs, otherwise estimated |
| Working days per month | 20 | Calendar |
| Input tokens per request | 1,500 | From gateway logs; system prompt and context dominate |
| Output tokens per request | 400 | From gateway logs |

From this follows: 200 active users, 120,000 requests per month, around 180 million input tokens and 48 million output tokens.

Input tokens almost always dominate the volume, output tokens almost always the cost — output is markedly more expensive than input at practically every vendor.

### 1.9.2 Variant A — external models only

At an assumed 3 US dollars per million input tokens and 15 US dollars per million output tokens, that gives around 540 dollars for input and 720 dollars for output, together some 1,260 dollars a month. On top comes operating the gateway, which runs on ordinary CPU nodes and hardly registers by comparison.

The decisive property of this variant: the cost is entirely variable. It doubles when usage doubles — and it falls to zero when nobody uses the platform.

### 1.9.3 Variant B — own inference

Here the cost is essentially fixed. An example calculation for a GPU server with two accelerator cards:

| Item | Amount per month | Basis |
|---|---|---|
| Hardware depreciation | around €700 | €25,000 over 36 months |
| Electricity | around €180 | approx. 1 kW continuous, €0.25/kWh |
| Data center space and network | around €100 | internal charging |
| Operational effort | around €1,500 | about 0.2 full-time equivalents |
| **Total** | **around €2,480** | |

The capacity of such a machine is orders of magnitude above the demand calculated above. That is precisely where the core of the problem lies: the fixed cost accrues whether the GPUs are 5 percent or 80 percent utilized.

### 1.9.4 The comparison and what it really says

At 120,000 requests a month the external variant is markedly cheaper. Break-even lies — under these assumptions — at about four times the usage, so around half a million requests a month. Beyond that, self-hosting wins, and increasingly clearly, because the hardware could carry a great deal more load. Section 14.14 runs this calculation in full and expresses break-even in terms of GPU utilization.

> **KEY POINT**
>
> Self-hosting pays off not above a certain number of users but above a certain **utilization**. A GPU used at one fifth is the most expensive variant of all — more expensive than any cloud API. That is the economic reason batching, sharing and scheduling take up so much room in this book.

And the real point: in many organizations the cost calculation is not the reason for self-hosting at all. The reason is data classification — for part of the requests the external option simply does not exist. The cost calculation then no longer decides the whether, only the sizing.

## 1.10 Typical false assumptions at the start

The following patterns turn up in almost every project. They are collected here because they almost all come from the same source: from carrying over assumptions that are right for classical workloads and no longer hold for inference.

**“Let’s buy some GPUs first and see.”** Hardware without usage data is almost always sized wrongly — usually too many cards and too little VRAM per card. VRAM per card determines which models can run at all; the number only determines throughput. The order is: measure first, then procure.

**“The model is too big, we need more GPUs.”** Frequently that is not true. Between “does not fit” and “more hardware” lie quantization, smaller model variants and a deliberately limited context length. Chapter 3 covers this chain systematically.

**“Autoscaling will take care of it.”** With inference it is not the application that scales but the model — and its start takes minutes. Autoscaling without a warm reserve produces outages at exactly the moment load arrives.

**“We’ll just log everything.”** Prompts can contain personal data, source code and trade secrets. Complete logging is therefore not an observability decision but a data protection one. It belongs before going live, not after.

**“A gateway is just a proxy.”** Functionally yes, operationally no. As soon as all AI usage runs through it, it is the most critical component of the platform — with corresponding requirements for high availability, state handling and maintenance windows.

**"GPU utilization at 100 percent means we are at the limit."** The value says only that compute cores are busy, not how efficiently. A server with poor batching reaches 100 percent at a fraction of the possible throughput.

**"The data scientists will tell us what they need."** They typically name model names, not resource profiles. Translating "we want this model" into VRAM demand, context limit, expected throughput and latency target is precisely the task of this role.

## 1.11 The environment: who else is involved

An AI platform touches more roles than an ordinary platform. Clarifying these interfaces early saves escalations later.

| Role | Needs from the platform | Delivers to the platform |
|---|---|---|
| Application teams | Stable, documented endpoints; quotas that fit the application | Load profiles, latency requirements, data classification |
| Data science | Being able to deploy models; reproducible environments | Model selection, resource profiles, quality criteria |
| Information security | Audit logs, access evidence, egress control | Classification scheme, approval criteria |
| Data protection | Information about storage and processing of prompts | Retention periods, purpose limitation, deletion rules |
| Procurement and controlling | Cost per team and application | Budgets, vendor contracts |
| Operations and on-call | Runbooks, informative alerts, clear escalation | Service hours, response times |

The "data protection" row is overlooked most often and causes late delays most often — usually at exactly the moment the platform is meant to go live.

The fourth row stands out. In classical platform work, data protection is rarely a daily interlocutor. On an AI platform it is, because the content of every request is potentially personal. Anyone who understands this coordination as part of the platform design rather than as a downstream approval builds the platform differently — and more correctly.

## 1.12 How this book is organized

From the fault lines a natural order follows. You cannot sensibly tune an inference server without knowing what the memory goes into. You cannot sensibly plan a gateway without knowing what stands behind it.

| Chapters | Topic | Answers the question |
|---|---|---|
| 2–3 | Inference and memory | What happens on the GPU, and what does it cost in VRAM? |
| 4–6 | GPU infrastructure | How does a GPU become a schedulable cluster resource? |
| 7–10 | Model serving | How does a GPU become a dependable model endpoint? |
| 11–13 | Enterprise layer | Who may do what, with which data, at what cost? |
| 14–16 | Operations | How is this kept running day to day, and how are changes delivered? |

| Chapters | Topic | Answers the question |
|---|---|---|
| 17 | Synthesis | What does the overall architecture look like at three orders of magnitude? |

This order deliberately does not follow the order in which the components stand from top to bottom in the architecture diagram. It follows the order in which they have to be understood: from the hardware upwards, because every higher layer makes assumptions about the one below.

### 1.12.1 What is runnable in the reference lab

Because the reference lab works with a single GPU, part of the material can be carried out in practice and part cannot. This boundary is stated explicitly in every chapter; here is the overview:

| Topic | In the lab | Note |
|---|---|---|
| GPU passthrough, driver stack | yes | Chapter 4, complete |
| GPU Operator on RKE2 | yes | Chapter 5, including the RKE2 particulars |
| Node pool separation, scheduling | yes | Chapter 5, realistic thanks to the topology |
| Time-slicing, MPS | yes | Chapter 6 |
| MIG | no | Not available on RTX cards; covered as concept and reference |
| vLLM, tuning, benchmarking | yes | Chapters 7–8, the core of the book |
| Tensor / pipeline parallelism | no | Needs several GPUs; optional cloud lab |
| KServe, canary, autoscaling | yes | Chapter 10 |
| LiteLLM, Keycloak, OpenBao | yes | Chapters 11–13, CPU workloads |
| Observability stack | yes | Chapter 14 |
| RAG path with vector DB | yes | Chapter 15, embedding model alongside the LLM |
| GitOps delivery | yes | Chapter 16 |

## 1.13 Interview questions

INTERVIEW QUESTION

*How do you distinguish your role from that of an ML engineer?*

**Weak** is any answer that lists responsibilities without justifying the cut.

**Good** is the distinction by artifact and runtime: the ML engineer is responsible for how a model comes about and gets better; the platform engineer is responsible for the environment in which it is usable for the organization.

**Senior** is what the interface between the two roles makes of the answer: the model artifact with its version, its resource requirements and its approval status. Anyone who can describe what that handover looks like concretely — registry, version reference in

the GitOps repo, resource profile, approval — shows they think in organizations and not only in technology.

**INTERVIEW QUESTION**

*Two thirds of an AI platform is supposedly classical platform engineering. What is the other third, then?*

**Good** is the list of the four new building blocks: GPU stack, inference server, serving abstraction, AI gateway.

**Senior** is the addition that it is not only about new components but about changed properties of familiar ones: indivisible resources, very large artifacts, utilization instead of availability as the headline metric, usage-proportional cost, and request content as a compliance object. Anyone naming the five fault lines has answered the question completely.

**INTERVIEW QUESTION**

*A board member asks why the company should run its own GPUs when OpenAI exists. What do you answer?*

**Weak** is a pure cost calculation — at moderate volume it comes out against self-hosting almost every time.

**Good** is the separation by data classification: for part of the data, external processing is simply not an option, regardless of price.

**Senior** is the hybrid answer with a condition: external models for the bulk of use cases, local models for the regulated minority — connected by a gateway that makes the decision from the data rather than leaving it to the applications. Together with an honest statement of the utilization level above which own hardware pays off, and the observation that a poorly utilized GPU is the most expensive variant of all.

**INTERVIEW QUESTION**

*What is the most important metric of an AI platform?*

**Weak** is "availability" — the answer from classical operations.

**Good** is GPU utilization, with the economic argument behind it.

**Senior** is the qualification: utilization in the sense of `nvidia-smi` only measures whether kernels are running, not how efficiently. What is informative is the combination of throughput in tokens per second, time to first token under load, KV cache occupancy and queue depth — and, for the business side, cost per thousand tokens compared to the market price.

**INTERVIEW QUESTION**

*Where would you start in an organization with no AI platform?*

> **Good** is the AI gateway: it creates visibility, access control and cost allocation immediately, without a single GPU having to be procured.
>
> **Senior** is the justification of the order — visibility before control, control before self-hosting — rounded off with the political point: the gateway is the component that can be justified fastest, because it solves a problem management already recognizes as a problem. Own GPUs come later, with dependable usage data from the gateway as the justification.

## 1.14 Summary

- The Enterprise AI Platform Engineer is responsible for the runtime environment for AI in the company: hardware, serving, access, governance, operations and cost — not for the domain quality of the model's answers.
- The role sits close to platform engineering, not to data science. Roughly two thirds of the knowledge required is already covered by anyone running Kubernetes in production.
- Four drivers create the demand: loss of control through shadow AI, cost that cannot be attributed, data sovereignty and vendor dependence.
- What is genuinely new is four building blocks — GPU stack, inference server, serving abstraction, AI gateway — and five structural fault lines: GPUs are not fungible, models are large and slow artifacts, utilization beats availability, cost arises per request, and the request content itself is the compliance object.
- The book is organized from the bottom up: understand inference, make GPUs available, serve models, regulate access and governance, secure operations.

### Sources and further reading


- NVIDIA: *GPU Operator Documentation* — reference for the complete GPU stack under Kubernetes. Basis for Chapters 5 and 6.
- vLLM: *Documentation* — architecture, engine arguments and distributed serving. Basis for Chapters 7 to 9.
- KServe: *Documentation* — InferenceService, serving runtimes and autoscaling. Basis for Chapter 10.
- LiteLLM: *Proxy Documentation* — gateway functions, virtual keys and budgets. Basis for Chapter 11.
- European Union: *Regulation on Artificial Intelligence (EU AI Act)* — relevant to the operator obligations in Chapter 13.

CHAPTER 2

# LLM Inference from an Infrastructure Perspective

| | |
|---|---|
| **GOAL** | Understand what happens on the GPU when a prompt arrives — far enough that memory demand, throughput and latency can be **calculated in advance** instead of measured afterwards. |
| **ASSUMES** | Chapter 1. No knowledge of neural networks. |
| **YOU WILL BE ABLE TO** | Calculate a model's KV cache demand from its configuration file, estimate decode speed from memory bandwidth, keep TTFT and throughput properly apart, and justify why batching is the central lever of every inference platform. |

**WRITTEN AGAINST**

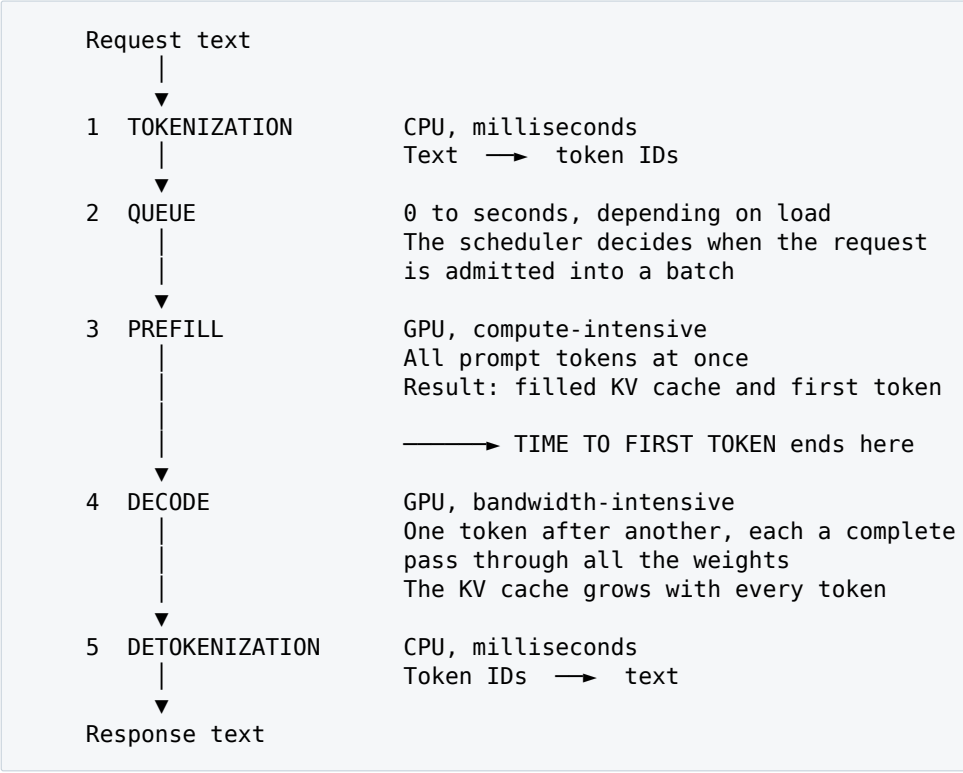
vLLM 0.8.x · Reference GPU RTX, 24 GiB · As of September 2026

This chapter is the foundation for everything that follows. Anyone who commands the calculations developed here can justify a GPU purchase, diagnose a capacity problem and answer any sizing question in an interview. Anyone who skips it will set vLLM parameters in Chapter 8 without knowing why.

What is explicitly not covered is how a transformer works. Of the model architecture, only what occupies memory and costs time is of interest here.

## 2.1 The life cycle of a request

From an operations perspective every inference request falls into five stages. Three of them are negligible, two determine everything.




The five stages of a request. Only prefill and decode cost GPU time.

**KEY POINT**

The entire operational mechanics of LLM inference follow from a single fact: **prefill and decode are two fundamentally different kinds of work.** Prefill is compute-bound and can be parallelized. Decode is memory-bandwidth-bound and is sequential in principle. Almost every optimization in this book addresses one or the other — rarely both.

## 2.2 Tokens

### 2.2.1 What a token is

A model does not process text but integers. The tokenizer splits text into units from the model's vocabulary — typically 32,000 to 200,000 entries — and maps them to IDs. These units are not words but frequent character sequences: whole short words, stems, endings, punctuation, spaces.

For platform operations exactly one property matters: **the number of tokens is not proportional to the number of words, and the ratio depends on the language.**

| Content | Tokens per word | Consequence |
|---|---|---|
| English prose | approx. 1.3 | The reference case vendors design their prices around |
| German prose | approx. 1.8 to 2.2 | Compounds and umlauts split into several tokens |
| Source code | approx. 2 to 3 | Identifiers, indentation and special characters |
| JSON and XML | approx. 3 and up | Structural characters dominate |

| Content | Tokens per word | Consequence |
|---|---|---|
| Tables and numbers | highly variable | Digit sequences are often split individually |

For the same statement, German therefore costs roughly one and a half to two times the tokens that English does. Anyone deriving a cost estimate from English-language benchmark figures and transferring it to a German-language use case underestimates systematically.

This line is almost always overlooked in cost estimates: the same application is noticeably more expensive in German than in English — through tokenization alone.

### 2.2.2 Input and output tokens

The distinction is not a pricing whim of the vendors but reflects technical reality. Input tokens are processed in prefill — all together, in one highly parallel pass. Output tokens are produced in decode — one at a time, in sequence, each with a complete pass through all the model weights.

An output token therefore costs a multiple of what an input token costs. With external vendors this shows up in prices that are typically three to five times higher for output. In self-hosted operation it shows up in the fact that output length decides the throughput of the entire platform.

> **KEY POINT**
>
> For capacity planning the rule of thumb is: the number of input tokens determines **memory demand** — through the KV cache. The number of output tokens determines **time** — through decode. Two different bottlenecks, two different levers.

## 2.3 Prefill: compute-bound

In prefill the GPU processes all prompt tokens in one pass. Because every input token is already known, the computation can be fully parallelized — exactly what a GPU is built for. Utilization of the compute units realistically reaches high values here.

The compute cost can be estimated well. A forward pass through a model with $P$ parameters costs approximately $2P$ floating-point operations per token — one multiplication and one addition per parameter. For a prompt of $T$ tokens, therefore:

$$\text{FLOPs}_{\text{prefill}} \approx 2 \cdot P \cdot T$$

For a model with 8 billion parameters and a prompt of 2,000 tokens that comes to roughly 32 trillion operations. A GPU of the reference class delivers in practice — not on the data sheet — roughly 80 to 120 trillion operations per second at 16-bit precision. Prefill therefore takes about 0.3 seconds.

The approximation ignores the cost of attention itself, which grows quadratically with sequence length. Up to a few thousand tokens the matrix multiplications of the weights dominate, and the approximation holds well.

That number is the lower bound of time to first token. Everything on top — queue, tokenization, network — is added to it.

> **KEY POINT**
>
> Prefill time grows **linearly** with prompt length. A prompt ten times as long means about ten times the wait until the first token. That is why applications with very large context — RAG with many embedded documents, for instance — have a noticeable latency problem that cannot be solved with more memory.

## 2.4 Decode: bandwidth-bound

Decode produces one token after another. Each new token requires the previous one — the steps cannot be parallelized.

What matters is what happens at each individual step: the GPU has to load **all the model weights** from video memory into the compute units in order to produce exactly one token. The compute cost for that is tiny — around $2P$ operations — but the data volume is the full model size.

The bottleneck is therefore not compute but memory bandwidth. And from that follows one of the most useful estimates in this book:

$$\text{Tokens per second} \approx \frac{\text{Memory bandwidth}}{\text{Model size in memory}}$$

| Model and format | Size | Calculated | Realistically measured |
|---|---|---|---|
| Llama 3.1 8B, FP16 | approx. 15 GiB | approx. 63 T/s | approx. 50–60 T/s |
| Llama 3.1 8B, AWQ 4-bit | approx. 5.7 GiB | approx. 175 T/s | approx. 110–140 T/s |
| Qwen 2.5 14B, AWQ 4-bit | approx. 9.5 GiB | approx. 105 T/s | approx. 70–90 T/s |
| Qwen 2.5 32B, AWQ 4-bit | approx. 19.5 GiB | approx. 51 T/s | approx. 30–45 T/s |

The basis is a memory bandwidth of around 1,000 GB/s, as offered by cards of the reference class. The measured values fall below it because the KV cache has to be read alongside the weights and because bandwidth is never fully exhausted. As an estimate for planning the calculation is nevertheless remarkably dependable.

> **KEY POINT**
>
> For a single request, **memory bandwidth** divided by **model size** determines output speed. A faster GPU with the same bandwidth gains practically nothing in decode. Quantization, by contrast, takes effect immediately because it shrinks the divisor — which makes it not only a memory measure but above all a speed measure.

> **CAUTION The most common mistake in GPU selection**
>
> Data sheets emphasize compute in TFLOPS. For decode — that is, for perceived response speed — this number is largely irrelevant. Compare memory bandwidth and VRAM capacity. A card with twice the compute but the same bandwidth delivers near-identical token rates for single requests.

## 2.5 The KV cache

### 2.5.1 Why it exists

At every decode step the model would in principle have to compute the relation of the new token to **all** preceding tokens. These intermediate results — key and value vectors — depend only on the tokens already processed, however, and do not change afterwards. They are therefore computed once and kept: in the KV cache.

Without it, the entire sequence so far would have to be reprocessed for every new token. The cost of a response would grow quadratically with its length rather than linearly. The KV cache thus trades compute time for memory — and that memory is the central capacity problem of every inference platform.

### 2.5.2 The formula

**KEY POINT KV cache demand**

$$\text{Bytes per token} = 2 \cdot L \cdot H_{\text{kv}} \cdot D \cdot B$$

with $L$ = number of layers, $H_{\text{kv}}$ = number of key-value heads, $D$ = dimension per head, $B$ = bytes per value (2 for FP16 or BF16). The factor 2 stands for key **and** value.

All four quantities can be read from the model's `config.json` — here the excerpt that matters for the calculation:

```
{
  "num_hidden_layers": 32,
  "num_key_value_heads": 8,
  "num_attention_heads": 32,
  "hidden_size": 4096,
  "torch_dtype": "bfloat16",
  "max_position_embeddings": 131072
}
```

Mapped to the symbols of the formula:

| Symbol | Field in `config.json` | Value in the example |
|---|---|---|
| $L$ | `num_hidden_layers` | 32 |
| $H_{\text{kv}}$ | `num_key_value_heads` | 8 |
| $D$ | `hidden_size` divided by `num_attention_heads` | 4096 / 32 = 128 |
| $B$ | `torch_dtype` — BF16 is 2 bytes | 2 |

For Llama 3.1 8B this gives:

$$2 \cdot 32 \cdot 8 \cdot 128 \cdot 2 \text{ bytes} = 131,072 \text{ bytes} = 128 \text{ KiB per token}$$

### 2.5.3 Why grouped-query attention matters so much

A practical mnemonic: 128 KiB per token means that 8,192 tokens occupy exactly 1 GiB. For Llama 3.1 8B the KV demand can be worked out in your head.

The formula contains $H_{\text{kv}}$, not the number of attention heads. In older models the two were the same. Modern models use **grouped-query attention**: several query heads share one key-value pair.

Llama 3.1 8B has 32 attention heads but only 8 KV heads. Without this optimization the demand would be 512 KiB per token instead of 128 — four times as much. The entire usable context of a server would be quartered.

| Model | $L$ | $H_{\text{kv}}$ | $D$ | per token | per 8,192 tokens |
|---|---|---|---|---|---|
| Qwen 2.5 0.5B | 24 | 2 | 64 | 12 KiB | 0.09 GiB |
| Qwen 2.5 7B | 28 | 4 | 128 | 56 KiB | 0.44 GiB |
| Llama 3.1 8B | 32 | 8 | 128 | 128 KiB | 1.00 GiB |

| Model | $L$ | $H_{kv}$ | $D$ | per token | per 8,192 tokens |
|---|---|---|---|---|---|
| Qwen 2.5 14B | 48 | 8 | 128 | 192 KiB | 1.50 GiB |
| Qwen 2.5 32B | 64 | 8 | 128 | 256 KiB | 2.00 GiB |
| Llama 3.1 70B | 80 | 8 | 128 | 320 KiB | 2.50 GiB |

What is remarkable about this table is that KV demand does **not** grow proportionally with parameter count. Llama 3.1 70B has almost nine times as many parameters as the 8B model but only 2.5 times the KV demand per token. Large models are expensive in weights and comparatively cheap in context — small models the other way round.

> **CAUTION The KV cache belongs to the sequence, not to the request**
>
> Demand follows the **entire** sequence length, that is prompt plus output produced so far. A request with 30,000 tokens of context occupies around 3.7 GiB on Llama 3.1 8B — continuously, until it finishes. A handful of such requests can occupy a server's cache completely and block everything else.

## 2.6 The VRAM balance sheet

A GPU's video memory divides into four items. Only one of them is variable, and that is precisely what sizing is about.

```
24 GiB VRAM
    ×  --gpu-memory-utilization 0.90
┌──────────────────────────────────────────────────────┐
│ claimed by the server:              approx. 21.6 GiB │
├──────────────────────────────────────────────────────┤
│ CUDA context, framework, buffers    approx.  0.8 GiB │ fixed
├──────────────────────────────────────────────────────┤
│ activations of the running batch    approx.  0.5 GiB │ nearly fixed
├──────────────────────────────────────────────────────┤
│                                                      │
│ MODEL WEIGHTS                       approx. 15.0 GiB │ fixed, but
│ Llama 3.1 8B in FP16                                 │ selectable via
│                                                      │ quantization
├──────────────────────────────────────────────────────┤
│                                                      │
│ KV CACHE                            approx.  5.3 GiB │ the remainder
│ = 43,000 tokens at 128 KiB per token                 │
│                                                      │
└──────────────────────────────────────────────────────┘
```

VRAM split: the KV cache gets whatever is left.

The KV cache is the residual item. It determines how many requests can be served at the same time and how long their context may be — and therefore the entire throughput.

### 2.6.1 A calculation you have to be able to run

Suppose a service is to serve 20 concurrent requests with 4,000 tokens of context each. The KV demand is:

$$20 \cdot 4{,}000 \cdot 128 \text{ KiB} = 10{,}240{,}000 \text{ KiB} \approx 9.8 \text{ GiB}$$

With 5.3 GiB of available cache that does not fit. There are four ways out, and all four have side effects:

| Option | Use when | Avoid when |
|---|---|---|
| Quantize the model | Reducing the weights to 4 bit: around 9 GiB more cache, so a budget of over 100,000 tokens. The most effective lever. | If response quality demonstrably suffers — which has to be assessed on the domain, not technically. |
| Limit context length | If the use case gets by with 2,000 tokens. Halves the demand immediately. | If the application has to process long documents. |
| Less concurrency | If requests may wait. Throughput is preserved, queueing delay rises. | If latency commitments apply or the queue grows without bound. |
| Smaller model | Qwen 2.5 7B needs 56 instead of 128 KiB per token and leaves more room for weights. | If the task genuinely requires the capabilities of the larger model. |
| Quantize the KV cache | FP8 for the cache halves its demand. Comparatively small effect on quality. | If the version in use does not support it for this model. |

**KEY POINT**

"More GPUs" is deliberately absent from this list. A second card does double the memory, but tensor parallelism costs communication and only pays off once the model itself no longer fits. For a capacity problem in the KV cache, quantization and a context limit are almost always the better answers. Chapter 9 covers when the exception applies.

## 2.7 Metrics

Without clean terms every discussion of inference performance is worthless. These five quantities have to be second nature.

| Metric | Meaning | What it depends on |
|---|---|---|
| TTFT | Time to first token: from admission to the first token emitted | Queue + prompt length + compute |
| TPOT / ITL | Time per output token, or inter-token latency: the gap between two output tokens | Memory bandwidth, model size, batch size |
| E2E latency | Total time until the complete response | TTFT + TPOT × output length |
| Throughput | Sum of output tokens per second across all requests | Batch size, KV cache size |
| Goodput | Share of requests that actually meet a latency commitment | All of the above together |

The relation is simple and is nevertheless confused constantly:

$$\text{E2E latency} = \text{TTFT} + \text{TPOT} \cdot (\text{output tokens} - 1)$$

At 0.3 seconds TTFT, 20 milliseconds per token and 500 output tokens that gives around 10.3 seconds. What the user perceives, however, depends hardly at all on this value and almost entirely on TTFT and TPOT — because modern interfaces stream the response. A response that starts after 0.3 seconds and continues fluently feels fast, even if it takes ten seconds.

> **KEY POINT**
>
> For latency commitments towards application teams, **TTFT** and **TPOT** are the right quantities, not total duration. Total duration depends on output length, and that is determined by the application, not by the platform. An SLO on E2E latency is therefore an SLO the platform cannot control.

Goodput is the most honest metric. A server can double its throughput by raising the batch size — and break every latency commitment doing so. Throughput then looks excellent and goodput falls to zero.

## 2.8 Batching — the central lever

### 2.8.1 Why it works at all

In decode, the entire model is read from memory at every step. This read is the same whether a token is computed for one request or for thirty at once. The bandwidth cost is paid once, the compute cost rises linearly — and compute is abundant during decode.

From this follows the most important effect in all of inference optimization: **throughput rises almost linearly with batch size while the latency of the individual request stays nearly constant** — until one of the two limits is reached.

| Batch | Tokens/s per request | Tokens/s total | Limiting factor |
|---|---|---|---|
| 1 | approx. 55 | approx. 55 | Memory bandwidth |
| 4 | approx. 53 | approx. 212 | Memory bandwidth |
| 16 | approx. 48 | approx. 768 | Memory bandwidth |
| 32 | approx. 42 | approx. 1,344 | Transition to compute |
| 64 | approx. 30 | approx. 1,920 | Compute and KV cache |
| 96 | – | – | KV cache exhausted, requests are queued |

The numbers are orders of magnitude for an 8B model on the reference hardware, not measurements — the lab at the end of this chapter produces your own. The pattern is general, however: total throughput rises almost proportionally at first, then flattens, and finally collapses once the KV cache is full.

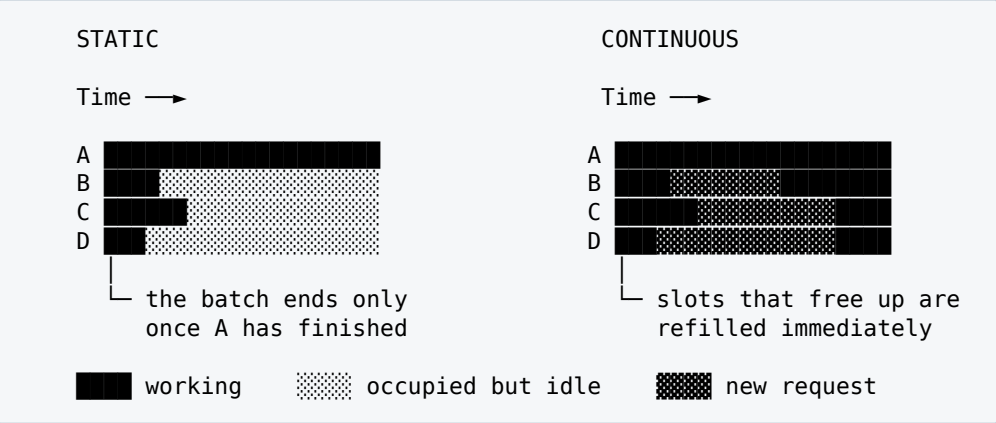
> **KEY POINT**
>
> An inference server without effective batching gives away an order of magnitude in throughput. That is the real reason a specialized server such as vLLM is so far superior to a simple script using Hugging Face Transformers — not better kernels, but better utilization. Chapter 7 shows how that works in detail.

### 2.8.2 Static, dynamic and continuous batching

| Method | How it works | Problem |
|---|---|---|
| Static | A fixed batch is formed, processed together, finished together. | Everyone waits for the longest response. Slots that free up stay unused. |
| Dynamic | The server waits a short window in order to collect several requests. | Better, but the batch stays rigid until the end. Addi- |

| Method | How it works | Problem |
|---|---|---|
| | | tional wait for the first request. |
| Continuous | After **every** decode step, finished requests leave the batch and new ones move in. | Requires memory management at block level — exactly what PagedAttention provides. |

The difference is considerable. With static batching the longest response in the batch determines how long every slot stays occupied. If one request produces 1,000 tokens and nineteen others 50 each, nineteen slots are idle for over 95 percent of the time.




Static versus continuous batching. Grey marks unused slots.

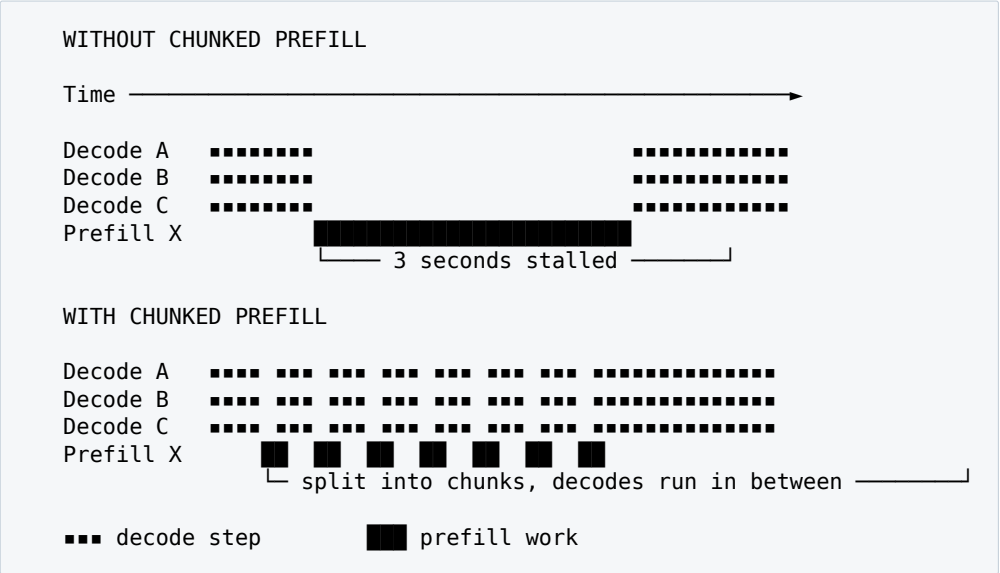
## 2.9 Prefill and decode compete for the same GPU

So far both phases have been considered separately. In live operation they happen at the same time — on the same card, with the same compute units. This produces the effect that causes most of the unexplained latency spikes in production.

### 2.9.1 The problem: one long request blocks all the others

A server is handling thirty requests in decode. Each step takes a few milliseconds, output runs fluently for all of them. Now a new request arrives with a prompt of 20,000 tokens.

Its prefill is a single, large, indivisible piece of work. As long as it runs, no decode step gets a turn. At around 0.3 seconds for 2,000 tokens, the prefill for 20,000 tokens takes roughly three seconds — and for three seconds the output of **all thirty** running requests stands still.




Head-of-line blocking: one large prefill stalls every decode.

In the metrics this looks as follows: TPOT jumps briefly to a multiple without utilization or memory occupancy standing out. Anyone looking only at averages sees nothing of it — only the 95th or 99th percentile makes the effect visible.

> **KEY POINT**
>
> This is why averages are misleading for inference latency. A server can have an excellent mean TPOT and still produce second-long stalls regularly. Latency SLOs for inference belong on percentiles, not on averages. Chapter 14 covers the histograms for that.

### 2.9.2 The countermeasures

| Measure | Effect | Price |
|---|---|---|
| Chunked prefill | The prefill is split into chunks; decode steps continue between the chunks. The blockage disappears. | The prefill itself takes slightly longer, TTFT of the new request rises a little. |
| Limit prompt length | Prevents the problem at the root. | Constrains use cases; has to be enforced in the gateway. |
| Separate instances by load profile | Applications with long prompts get their own endpoints; interactive use stays unaffected. | More instances, worse overall utilization. |
| Prefill/decode disaggregation | Prefill and decode run on **different** GPUs; the KV cache is transferred. Both phases can be scaled independently. | Considerably higher complexity; only pays off on large installations. Chapter 9. |

Chunked prefill is available in current versions of the common servers and in many cases already the default. It is the first measure to check when TPOT outliers appear.

### 2.9.3 The special case: mixed load profiles

The conflict sharpens when very different use cases share the same endpoint:

| Use case | Prompt | Output | Character |
|---|---|---|---|
| Chat assistant | short | medium | decode-dominated, latency-sensitive |

| Use case | Prompt | Output | Character |
|---|---|---|---|
| RAG with documents | very long | medium | prefill-dominated |
| Classification | medium | very short | prefill-dominated, bulk operation |
| Code generation | long | long | both, high KV demand |
| Summarization | very long | short | almost pure prefill |

A single endpoint serving all five profiles will be well configured for none of them. Separation by load profile — through dedicated instances with dedicated parameters, routed by the gateway — is one of the most effective architectural decisions of a mature platform.

**KEY POINT**

Before you size an inference endpoint, establish the load profile: how long is the prompt, how long the response, how many requests at once, and how sensitive is the application to latency? Without those four figures every configuration is guesswork.

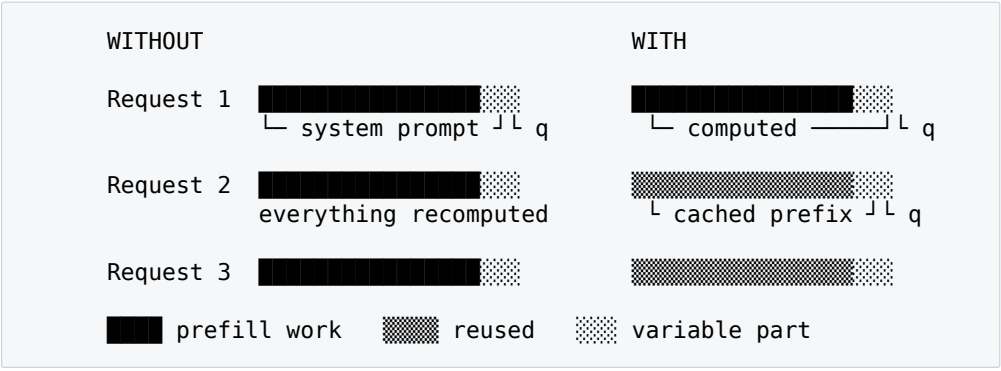
This table is a good opening for a conversation with application teams. The question "how long are your prompts and your responses?" yields more for sizing than any statement about user numbers.

## 2.10 Shared prefixes: the underrated lever

In enterprise applications, requests look very much alike. Almost every one carries the same system prompt: role description, rules of behaviour, output format, tool definitions. In practice that is frequently 500 to 2,000 tokens — ahead of every single user question.

Without a countermeasure the server recomputes this identical beginning for **every** request. With a system prompt of 1,500 tokens and a user question of 200 tokens, almost nine tenths of all prefill work goes into the same text over and over.

**Prefix caching** solves this: the computed key-value blocks of the shared beginning stay in the cache and are reused by subsequent requests. Prefill starts only where the requests actually differ.




Without and with prefix caching, same system prompt

The effect is considerable: with the figures above, prefill work drops by around 88 percent from the second request onwards, and TTFT falls accordingly.

**KEY POINT The order within the prompt decides**

Only a **shared beginning** can be reused. As soon as two requests differ at any point, nothing beyond that point is reusable. From this follows a rule that platform operators should pass on to their application teams:

**Static first, variable last.** System prompt, tool definitions and format requirements at the beginning; retrieved documents, user data and the actual question at the end. A timestamp or a user name at the start of the prompt renders prefix caching completely ineffective — a mistake that occurs frequently in RAG applications and whose cause is rarely looked for.

The price is memory: cached prefixes occupy blocks in the KV cache that would otherwise be available to running requests. With a few frequently used system prompts the trade is clearly favourable. With very many different prefixes the cache displaces itself and the benefit disappears. Chapter 7 covers the eviction strategy in detail.

## 2.11 Output length and sampling from an operations perspective

The parameters with which applications steer the output look like domain settings. From a platform perspective several of them are directly capacity-relevant.

| Parameter | Effect on the domain | Effect on operations |
|---|---|---|
| `max_tokens` | Limits response length | **The single most effective cost brake.** Without an upper bound a single request can occupy a batch slot for arbitrarily long. |
| `temperature` | Steers randomness of selection | Hardly any effect on compute. At 0 results are reproducible — important for tests and for response caching. |
| `top_p`, `top_k` | Limit the candidate space | Negligible. |
| `n` | Produces several response variants | Multiplies decode work directly. `n = 4` means four times the cost. |
| `stop` | Ends output at defined character sequences | Saves decode time when set cleanly — often more than any server optimization. |
| `stream` | Delivers tokens one at a time | Changes nothing about compute load, but everything about perceived speed. |

**CAUTION A missing upper bound is a capacity risk**

A client without `max_tokens` can produce a response that runs to the context limit. At 4,096 tokens of context that is, in the worst case, over 4,000 decode steps for a single request — while permanently occupying a batch slot and the associated KV memory. A server-side upper bound in the gateway is among the first protective measures of any platform. Chapter 11 shows how.

## 2.12 Embedding inference: same hardware, different load shape

For RAG applications an embedding model is operated alongside the language model. It deserves its own section here because its load profile looks different in almost every respect — and because it frequently has to share the same GPU with the language model.

| Property | Language model | Embedding model |
|---|---|---|
| Phases | prefill and decode | prefill only |
| KV cache | grows with every response | does not arise at all |
| Bottleneck | memory bandwidth in decode | compute |
| Typical size | 5 to 20 GiB | under 1 GiB |
| Batching | continuous, complex | simple, very effective |
| Response time | seconds | milliseconds |

Because no decode takes place and no KV cache arises, embedding inference scales far more kindly: large batches are unproblematic, and throughput is orders of magnitude higher — thousands of short texts per second are achievable.

The operational consequence is shared memory. An embedding model of just under 1 GiB takes around 8,000 tokens of KV cache away from the language model. That is bearable, but it has to appear in the balance sheet — and it argues for running embedding services on a CPU node where the latency requirement permits.

In RAG systems the bottleneck almost never arises when embedding the query, but in the one-off processing of the document corpus. Chapter 15 treats this ingestion load separately from query operation.

## 2.13 Context length and its price

Models state a maximum context length — Llama 3.1 for instance 131,072 tokens. That number is a property of the model, not an operational commitment. What a server can actually offer follows from its KV cache.

At 5.3 GiB of cache and 128 KiB per token the total token budget is around 43,000 tokens — across **all** concurrent requests together. The full model context of 131,072 tokens therefore does not even fit for a single request.

> **CAUTION The classic first mistake when starting vLLM**
>
> Without a context limit set, vLLM tries to serve the model's declared maximum length, finds that the KV cache is not sufficient for it, and aborts. The message names, in substance, the model's maximum sequence length and the number of tokens that actually fit into the cache, and suggests lowering `--max-model-len` or raising `--gpu-memory-utilization`. The exact wording changes between versions; the pattern stays.
>
> The right reaction is not to turn `--gpu-memory-utilization` blindly, but to align the context limit deliberately with the use case. Chapter 8 covers both parameters in detail.

Context length is thus a **capacity decision**, not a configuration formality. It directly determines how many requests can run at the same time:

$$\text{Concurrent requests} \approx \frac{\text{KV cache in KiB}}{\text{context length} \cdot \text{KiB per token}}$$

| Context limit | Concurrent requests | Suitable for |
|---|---|---|
| 2,048 tokens | approx. 21 | Chat, short tasks, classification |
| 4,096 tokens | approx. 10 | General operation |
| 8,192 tokens | approx. 5 | RAG with few documents |
| 32,768 tokens | approx. 1 | Long documents — effectively single-request operation |

## 2.14 From user count to GPU count

With that, all the pieces are in place to answer the question asked sooner or later in every platform conversation: **how many GPUs do we need?** The calculation runs in five steps.

### 2.14.1 Step 1 — request rate

The usage assumption becomes a rate. For the 200 active users from Chapter 1 with 30 requests each over an eight-hour working day:

$$\frac{200 \cdot 30}{8 \cdot 3600\ \text{s}} \approx 0.21\ \text{requests per second}$$

Usage is not evenly distributed, however. A peak factor of 3 to 5 over the daily mean is realistic for interactive use — mornings and the time after meetings dominate. With a factor of 3:

$$\text{Peak load} \approx 0.63\ \text{requests per second}$$

### 2.14.2 Step 2 — token demand

At 400 output tokens per request:

$$0.63 \cdot 400 \approx 250\ \text{output tokens per second}$$

That is the quantity a GPU has to deliver — not the request rate.

### 2.14.3 Step 3 — throughput per GPU

From the batching section: an 8B model on the reference hardware delivers roughly 1,000 to 1,400 output tokens per second at medium batch size. On paper a single card covers the demand several times over.

### 2.14.4 Step 4 — cross-check against the KV cache

Throughput alone is not enough; the requests also have to fit into memory at the same time. By Little's law, for the number of requests concurrently in the system:

$$\text{Requests in system} = \text{rate} \cdot \text{residence time}$$

At 0.63 requests per second and a mean handling time of 8 seconds that is around 5 concurrent requests. At 4,096 tokens of context and 128 KiB per token:

$$5 \cdot 4,096 \cdot 128\ \text{KiB} \approx 2.5\ \text{GiB}$$

That fits comfortably into the available 5.3 GiB. The KV cache is not the bottleneck here.

### 2.14.5 Step 5 — reserve and availability

Capacity that suffices on paper is not the capacity to procure. Three surcharges are added:

| Surcharge | Factor | Reason |
|---|---|---|
| Growth | 1.5 to 2 | Usage regularly grows faster after launch than planned |
| Rollout reserve | +1 instance | During model changes two versions run in parallel briefly |
| Failure reserve | +1 node | Without a second GPU node, maintenance means an outage |

**KEY POINT**

The result of this calculation is almost always that the **capacity** for the planned demand is reached with astonishingly little hardware — and that actual procurement is determined by **availability** and **model size**, not by user count. Anyone planning a platform with two GPU nodes does so for maintainability and fault tolerance, not because of throughput.

**CAUTION Where this calculation breaks down**

It holds for interactive use with short responses. Two cases break it: **batch processing** — classifying millions of documents, say — produces sustained load without peaks and is limited by raw throughput. **Agentic applications** produce ten to a hundred model calls per user action instead of one; here the multiplier between user action and request rate is the decisive and usually unknown quantity.

## 2.15 Why measurements scatter

Anyone checking the calculations of this chapter against their own hardware will see diverging numbers. That is normal. The following causes explain almost every deviation and should be ruled out for every measurement.

**No warm-up** The first requests after startup are markedly slower — kernels are compiled, caches fill, execution graphs are built. Discard the first runs as a matter of principle.

**Different tokenization** "100 tokens" means a different amount of text for two models. Comparisons between models have to run on tokens, not on characters or words.

**Varying output length** At `temperature` above zero the same prompt produces responses of different lengths. For measurements set `temperature` to 0 and fix `max_tokens`, otherwise you measure output length and not speed.

**Prefix caching acts unnoticed** The second run of the same prompt is drastically faster. For prefill measurements the prompts have to differ.

**Thermal limiting** Consumer cards throttle under sustained load. A run over ten minutes yields different values than a run over ten seconds. Watch clock frequency and temperature alongside.

**Foreign load on the same card** A forgotten process occupies memory and compute time. Check the occupancy state before every measurement.

**KEY POINT**

A measurement without recorded boundary conditions — model, quantization, context limit, batch size, prompt length, output length, server version — is not reproducible and therefore worthless. That holds especially for numbers meant to justify a purchase later on.

## 2.16 Lab: making inference measurable

**LAB Checking the theory on your own GPU**

*Goal: Separate prefill and decode behaviour, verify the bandwidth formula and make the batching effect visible.*

**Prerequisite:** GPU node reachable, Python environment with vLLM. Full cluster operation follows in Chapter 8 — here a direct start on the GPU host is enough.

**Step 1 — start the server.** Deliberately with a limited context length so that the start succeeds:

```
vllm serve Qwen/Qwen2.5-7B-Instruct \
    --max-model-len 4096 \
    --gpu-memory-utilization 0.90 \
    --port 8000
```

**Expected output:** In its startup message vLLM names the size of the KV cache in tokens or GiB. Note this value — it is the basis of all the calculations that follow. Check it against the formula from Section 2.5: Qwen 2.5 7B occupies 56 KiB per token.

**Step 2 — decode rate for a single request.** A request with a short prompt and a long output isolates decode:

```
curl -s http://localhost:8000/v1/completions \
  -H 'Content-Type: application/json' \
  -d '{"model":"Qwen/Qwen2.5-7B-Instruct",
       "prompt":"Count from 1 to 200.",
       "max_tokens":300, "temperature":0}' \
  | jq .usage
```

Measure the wall-clock time with `time`. Divide the tokens produced by the duration. **Expectation:** the value should be in the order of bandwidth divided by model size — so for a 7B model in FP16 roughly 60 to 70 tokens per second.

**Step 3 — make prefill visible.** Repeat the call with a very long prompt and `"max_tokens": 1`. Decode then falls away almost entirely, and the measured time is essentially the prefill. Raise the prompt length in stages — 256, 1,024 and 3,072 tokens, say — and plot the times. **Expectation:** an approximately linear curve.

**Step 4 — the batching effect.** Send 1, 4, 16 and 32 requests at the same time, each with identical output length:

```
M=Qwen/Qwen2.5-7B-Instruct
BODY='{"model":"'$M'","prompt":"Tell a story.",'
BODY=$BODY'"max_tokens":200,"temperature":0}'
```

```
for n in 1 4 16 32; do
  start=$(date +%s.%N)
  for i in $(seq 1 $n); do
    curl -s http://localhost:8000/v1/completions \
      -H 'Content-Type: application/json' \
      -d "$BODY" >/dev/null &
  done
  wait
  d=$(echo "$(date +%s.%N) - $start" | bc)
  echo "n=$n  ${d}s  $(echo "$n*200/$d" | bc) tokens/s"
done
```

**Expectation:** total throughput rises markedly while the duration of the individual request increases only moderately. That is precisely the batching effect from Section 2.8.

**Step 5 — read the metrics.** During a load phase:

```
curl -s http://localhost:8000/metrics | grep -E \
  'num_requests_|gpu_cache_usage|time_to_first_token'
```

Watch how the number of waiting requests and KV cache occupancy rise with concurrency. These two values are the most important signals for autoscaling — Chapters 10 and 14 build on them.

**Result:** You have verified the three central claims of this chapter on your own hardware: prefill grows linearly with prompt length, decode is bound to bandwidth, and batching raises throughput without degrading single-request latency correspondingly.

## 2.17 Operations and troubleshooting

| Symptom | Cause | Diagnosis | Fix |
|---|---|---|---|
| Server does not start, message about maximum sequence length | Declared model context does not fit into the available KV cache | Compute model size and `max_position_embeddings` against free VRAM | Align `--max-model-len` with the use case; consider a quantized model |
| Responses very slow, GPU utilization low | No effective batching; requests run one after another | Check `num_requests_running` — it sits permanently at 1 | Raise concurrency; check whether the client serializes |
| Throughput collapses as load rises | KV cache exhausted, requests are preempted and recomputed | `gpu_cache_usage_perc` near 1.0 with a growing queue | Lower the context limit, quantize the model or limit concurrency |
| TTFT high, TPOT normal | Long prompts or queueing delay | Plot TTFT against prompt length; check `num_requests_waiting` | Enable chunked prefill; add capacity; shorten prompts |
| TPOT high, TTFT normal | Batch too large or model too large for the bandwidth | Check tokens per second against bandwidth divided by model size | Quantize; limit maximum batch size |

| Symptom | Cause | Diagnosis | Fix |
|---|---|---|---|
| Memory errors only under load, not at startup | Weights fit, but KV cache and activations grow with concurrency | Watch occupancy during a load test | Balance `--max-num-seqs` against the memory share |

## 2.18 Interview questions

**INTERVIEW QUESTION**

*A model needs 40 GB of VRAM. You have a card with 24 GB. What do you do?*

**Weak** is "more GPUs" as the first and only answer.

**Good** is the ordered list: quantization, smaller model variant, limit context length, several GPUs with tensor parallelism, CPU offloading as a last resort.

**Senior** is what the ordering and its justification make of the answer: quantization first, because it both saves memory and accelerates decode — model size sits in the denominator of the bandwidth formula. Tensor parallelism last, because it introduces communication cost and only pays off once the weights themselves do not fit. CPU offloading only as an emergency measure, because PCIe bandwidth is more than an order of magnitude below HBM bandwidth. Plus the counter-question of whether 40 GB means the weights alone or includes the KV cache — that changes the answer entirely.

**INTERVIEW QUESTION**

*Why does an LLM need so much GPU memory?*

**Good** is the split into weights and KV cache.

**Senior** is the complete balance sheet — weights, KV cache, activations, CUDA context — with the observation that the weights are fixed and the KV cache is the variable remainder that determines concurrency and context length. Anyone who can name the formula $2LH_{\text{kv}}DB$ and work it through on a model stands out clearly.

**INTERVIEW QUESTION**

*What is the difference between prefill and decode, and why should I care?*

**Good** is: prefill processes the prompt in parallel, decode produces tokens sequentially.

**Senior** is the operational consequence: prefill is compute-bound and determines TTFT, decode is bandwidth-bound and determines TPOT. From this it follows that a latency problem has entirely different causes depending on its shape — and that both phases compete for the same GPU. Anyone naming chunked prefill or prefill/decode disaggregation as the answer shows they follow current developments.

**INTERVIEW QUESTION**

*How many concurrent users can one GPU handle?*

**Weak** is any concrete number without a counter-question.

**Good** is the observation that it depends on model, quantization, context length and output length.

**Senior** is doing the calculation on the spot: KV cache divided by context length times bytes per token gives the number of concurrent sequences; throughput divided by tokens per request gives the number of requests per second. Rounded off with the note that "concurrent users" and "concurrent requests" are not the same thing — a user in a chat perhaps produces one request per minute.

**INTERVIEW QUESTION**

*Why is quantization not only a memory measure?*

**Good** is: it reduces model size and makes room for the KV cache.

**Senior** is the second effect: because decode is bandwidth-bound and model size sits in the denominator, quantization directly raises output speed — a 4-bit model is not only smaller but roughly two to three times faster in decode. The price is a loss of quality that has to be assessed on the domain. Chapter 3 covers both.

## 2.19 Summary

- A request falls into prefill and decode. Prefill is compute-bound, grows linearly with prompt length and determines TTFT. Decode is bandwidth-bound, sequential, and determines TPOT.
- The decode speed of a single request can be estimated as memory bandwidth divided by model size. This formula holds for planning and for fault-finding.
- The KV cache needs $2LH_{\text{kv}}DB$ bytes per token. All values are in the model's `config.json`. For Llama 3.1 8B it is 128 KiB per token, that is 1 GiB per 8,192 tokens.
- VRAM divides into weights, KV cache, activations and context. The weights are fixed, the KV cache is the remainder — and it determines concurrency and context length.
- Context length is a capacity decision, not a formality. The declared model context is not an operational commitment.
- Batching is the most effective lever for throughput, because the bandwidth cost of reading the weights is paid only once per decode step. Continuous batching requires block management of the cache.
- For latency commitments, TTFT and TPOT are the right quantities. Total duration depends on output length, which the application determines.

### Sources and further reading


- vLLM: *Documentation — Engine Arguments* and *Metrics*. Reference for `max-model-len`, `gpu-memory-utilization` and the metrics used in the lab.
- Kwon et al.: *Efficient Memory Management for Large Language Model Serving with PagedAttention*. The work behind block management of the KV cache; foundation for Chapter 7.
- Ainslie et al.: *GQA — Training Generalized Multi-Query Transformer Models*. Background on reducing the number of key-value heads.

- The `config.json` of the models in use, on Hugging Face — the most reliable source for $L$, $H_{\text{kv}}$ and $D$.