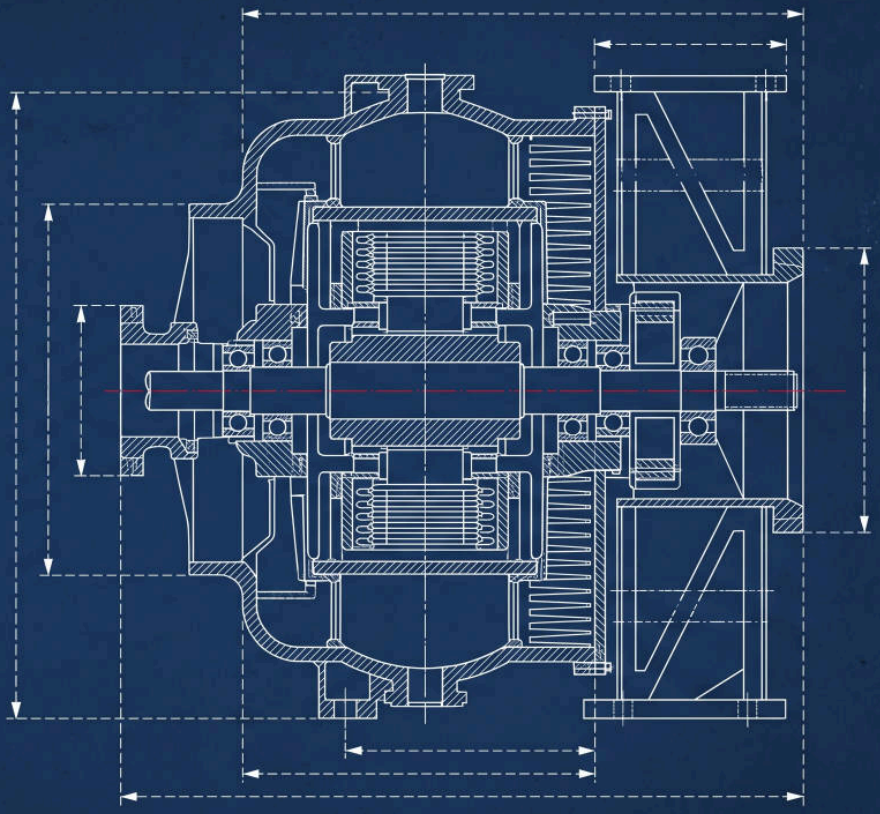

CHAPTER FOUR

Running Models Locally

SECTIONS 4.1 TO 4.11

Self hosting is the decision most often made for the wrong reason and abandoned for the right one. This chapter ends with the arithmetic that tells you which side of the break-even you are on.

CHAPTER FOUR

Contents

Self hosting is the decision most often made for the wrong reason and abandoned for the right one. The economics are genuinely favourable at a specific scale and genuinely terrible on either side of it, and this chapter ends with the arithmetic that tells you which side you are on.

4.1 Ollama Overview: Local LLM Runtime Engine

Ollama is a convenience layer over llama.cpp. It pulls quantised weights, manages a model registry, and exposes an HTTP API. It is excellent for development and for single-user workloads, and it is the wrong tool for multi-user serving, for reasons that are structural rather than a matter of tuning.

Three of its defaults are wrong for anything beyond a laptop, and **two of them fail silently**, which is worse than failing.

Default one: models unload on idle

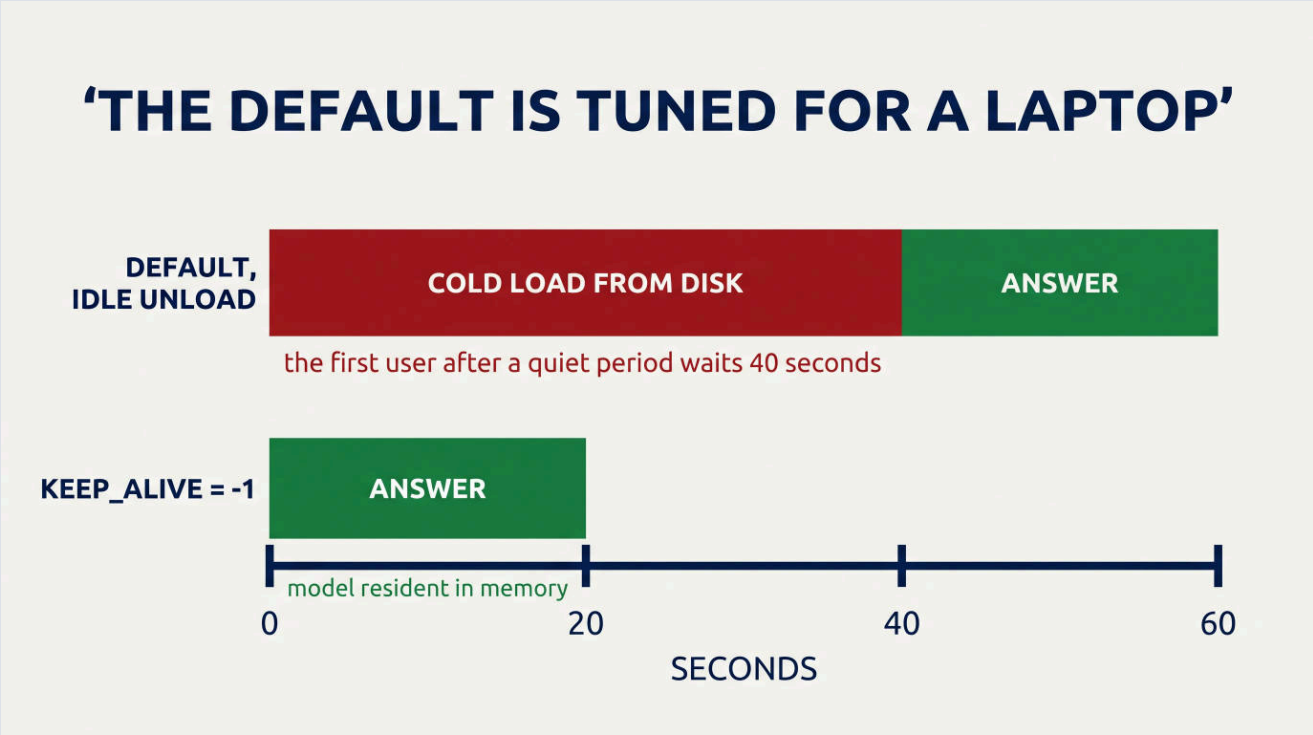

Figure 4.1 The default keeps a model resident for five minutes after the last request. The next user pays for the reload.

This is invisible in development, where you are always warm, and it is the single most common complaint about self hosted deployments. A product with traffic gaps, which is every product with a working day, serves a thirty to sixty second wait to the first user after each quiet period.

```
# compose.yaml. -1 means never unload.
environment:
  OLLAMA_KEEP_ALIVE: "-1"
  OLLAMA_MAX_LOADED_MODELS: "1"   # loading two 8B models on one 24 GiB card
  OLLAMA_NUM_PARALLEL: "4"        # will succeed, then OOM under load
  OLLAMA_MAX_QUEUE: "64"          # bounded: reject rather than accumulate
```

Default two: concurrency is limited and the batching is basic

Ollama processes a small number of parallel sequences. It does not do continuous batching in the sense section 4.5 develops, which is where the large throughput multipliers live. For an internal tool with five users this does not matter. For a product it caps you an order of magnitude below what the same hardware can deliver.

Default three: the context window is small, and truncation is silent

This is the dangerous one. Ollama historically defaults to a modest context window regardless of what the model supports, and content beyond it is *truncated without an error*.

Follow the consequence for a retrieval system. Your pipeline assembles twelve thousand tokens of carefully reranked evidence. The runtime is configured for four thousand. The model receives a third of the evidence, produces a fluent and plausible answer, and nothing anywhere in your system knows that two thirds of the input was discarded. Section

2.1 established that fluency is uncorrelated with correctness; this is the mechanism by which that becomes a production incident.

```
# src/docassist/serving/runtime_check.py
async def assert_runtime_ready(client, host, model, *, required_ctx: int):
    """Raised at STARTUP, not discovered in a quality review three months on."""
    r = await client.post(f"{host}/api/show", json={"model": model}, timeout=30.0)
    r.raise_for_status()
    info = r.json()

    ctx = _find_context_length(info)
    if ctx is None:
        raise RuntimeMisconfigured(f"cannot determine context length for {model}")
    if ctx < required_ctx:
        raise RuntimeMisconfigured(
            f"{model} context is {ctx} but the pipeline assembles up to "
            f"{required_ctx} tokens. Silent truncation would occur."
        )
    return {"model": model, "context_length": ctx}
```

Call it in the FastAPI lifespan, with `required_ctx` taken from the `ContextBudget` of section 2.10. The budget already knows its own ceiling, so the two numbers cannot drift apart. A service that refuses to start is a far better outcome than a service that answers from a third of the evidence.

> ALWAYS SET NUM_CTX EXPLICITLY, AND ASSERT ON IT
>
> Passing `num_ctx` per request is not sufficient on its own, because a value larger than the loaded model's configuration is silently clamped. The pattern is: set it in the request, verify it at startup, and fail loudly if the runtime cannot honour it.

4.2 Running Ollama Models with Docker

The compose service from section 1.3, with the settings that matter for anything beyond a laptop.

```
ollama:
  image: ollama/ollama:0.5.4                  # pinned, like any dependency
  volumes: ["ollama_models:/root/.ollama"]    # NEVER bake weights in the image
  environment:
    OLLAMA_KEEP_ALIVE: "-1"
    OLLAMA_NUM_PARALLEL: "4"
    OLLAMA_MAX_QUEUE: "64"
  deploy:
    resources:
      limits: {memory: 12G}                   # or it takes the host down
      reservations:
        devices: [{driver: nvidia, count: 1, capabilities: [gpu]}]
  healthcheck:
    test: ["CMD-SHELL", "ollama list || exit 1"]
    interval: 10s
    start_period: 120s                        # the first pull is slow
```

Two operational details are worth stating because both cause avoidable pain.

Pre-pull the model in an init step, not on first request. Otherwise your first user pays for a multi-gigabyte download, which is the cold start of section 4.1 one layer up and an order of magnitude worse.

The memory limit is the line that protects your laptop and your database. A local model runtime will consume all available memory and take the vector database down with it. The symptom without a limit is a machine that freezes hard; with a limit it is a single container that restarts, which is survivable and diagnosable.

The GPU reservation in that file is optional. Without it Ollama runs on CPU, which is the path section 1.1 promised: expect single-digit tokens per second on an 8B model rather than tens, slow but sufficient for every exercise in this chapter and the rest of the book.

4.3 Configuring OpenWebUI with an Ollama Backend

OpenWebUI is a chat front end that speaks to Ollama or any OpenAI-compatible endpoint. Its architectural value is not the interface. It is that it gives non-engineers a way to exercise your models, which is how you collect the real questions that become eval cases in Chapter 14.

Two configuration points, both about where it points.

Point it at your gateway, not directly at Ollama, once the gateway exists. Internal usage then flows through the same routing, tracing and budget path as production traffic. Otherwise your internal usage is invisible in exactly the

dashboards you built to see usage, and the questions your colleagues ask never reach your eval set.

Treat it as an authenticated internal service. An unauthenticated chat interface with access to your models is an open proxy for anyone who finds the port, and it bills to your account.

4.4 FastAPI Environment Setup and Dependencies

FastAPI is the API tier for the rest of the book. Three setup decisions have architectural consequences and are usually made by copying a tutorial.

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    s = settings()
    # (1) ONE client for the process lifetime.
    app.state.http = httpx.AsyncClient(
        timeout=httpx.Timeout(s.request_timeout_s, connect=s.connect_timeout_s),
        limits=httpx.Limits(max_connections=200, max_keepalive_connections=50),
    )
    app.state.tracer = Tracer()
    app.state.registry = PromptRegistry("prompts")
    app.state.gateway = LLMGateway(settings=s, client=app.state.http,
                                   secrets=SecretProvider(), tracer=app.state.tracer)
    yield
    await app.state.http.aclose()


# (2) Health endpoints that do NOT depend on the model.
@app.get("/alive")
async def alive() -> dict:
    """Process health ONLY. If this depends on the model, a slow model
    triggers a restart loop and the outage becomes self-inflicted."""
    return {"ok": True}

@app.get("/ready")
async def ready(request: Request) -> dict:
    """Capacity. The load balancer reads this one."""
    gw = request.app.state.gateway
    return {"ok": gw.breaker.allow(), "circuit_open": not gw.breaker.allow()}


# (3) Never do blocking work in an async handler.
@app.post("/ingest")
async def ingest(file: UploadFile):
    return await asyncio.to_thread(parse_pdf_sync, await file.read())
```

(1) Creating an HTTP client per request defeats connection pooling and leaks file descriptors. It surfaces as mysterious timeouts under load, hours after the deploy, and it is a one-line fix that nobody finds because the symptom points elsewhere.

(2) This is the direct response to the cascade in section 3.6. Separating liveness from readiness is what stops the orchestrator killing a healthy pod because a downstream model is slow. `/alive` must never touch anything that can block.

(3) One synchronous PDF parse in an async endpoint stalls *every* concurrent request on that worker, because the event loop is single threaded. This is the async-specific version of the worker exhaustion problem, and it is easy to introduce accidentally by calling a library that does file or CPU work.

4.5 Integrating Ollama with FastAPI

GOOD

The promoted laptop

```
@app.post("/answer")
async def answer(req: AnswerRequest):
    r = await httpx.AsyncClient().post(          # new client per request
        f"{OLLAMA_HOST}/api/generate",
        json={"model": "llama3.1:8b", "prompt": req.prompt, "stream": False})
    return {"answer": r.json()["response"]}
```

Four lines, works on a laptop, and structurally incapable of serving two users at once. What breaks, in order:

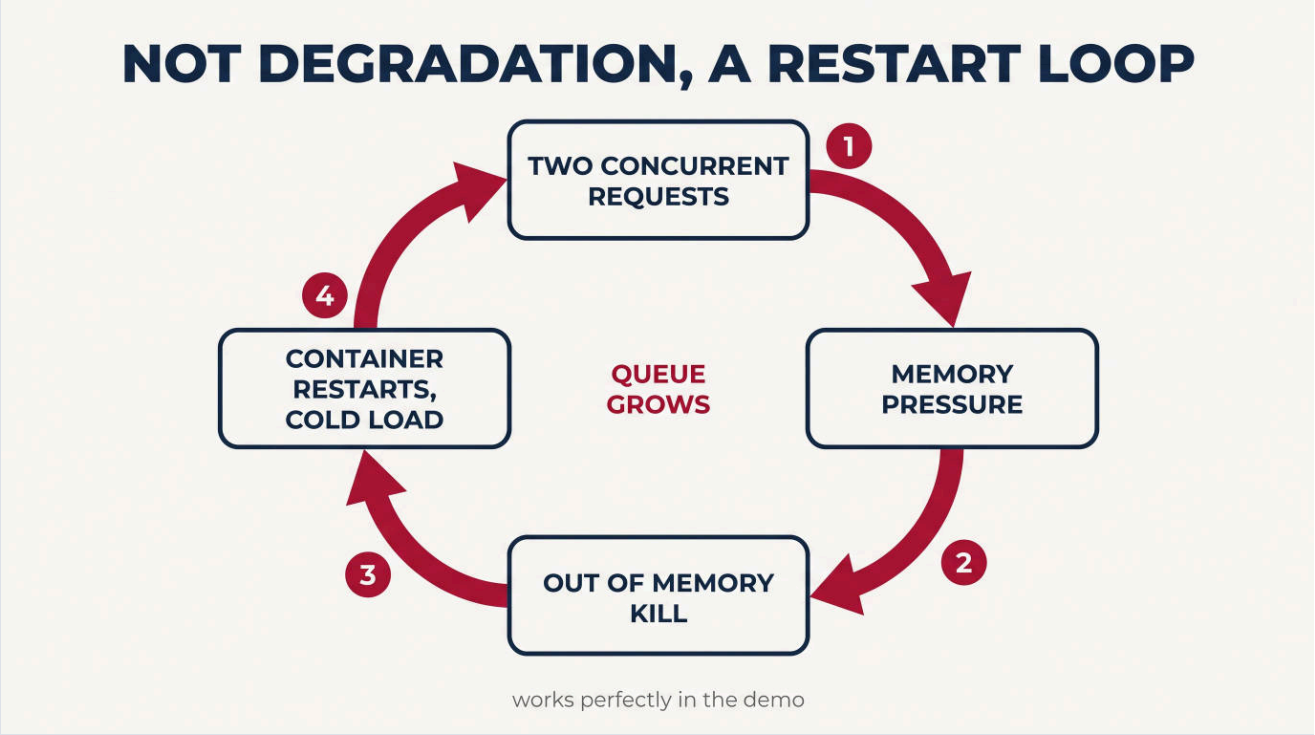

Figure 4.2 Failure here is not graceful degradation. It is a loop that gets worse each time round.

The loop is worth following because it explains why the obvious mitigation makes things worse. Two concurrent requests cause memory pressure. The kernel out-of-memory killer terminates the container. The container restarts, which forces a cold load from section 4.1. During that load the queue of waiting requests grows. When the model finally becomes resident, it is immediately hit by a larger burst than the one that killed it, so it dies again.

Adding memory delays the first kill and does nothing about the loop. The fix is admission control, which is the next tier.

Two further defects in those four lines. **A new HTTP client per request**, per section 4.4. And `stream: False`, which on a self hosted small model producing thirty tokens per second means a four hundred token answer is thirteen seconds of blank screen, when it could have been half a second to first token.

The model names in this chapter's listings, `llama3.1:8b` here and `gemma-2-9b-it` in section 4.6, are illustrative snapshots from the time of writing, in the same way as the pins of section 1.2: expect better weights at the same sizes by the time you run them, and read every argument as being about a size class rather than a name.

> WHEN THIS IS NEVERTHELESS CORRECT
>
> Internal tools with single-digit concurrent users, and any batch job where latency does not matter. It is a genuinely good architecture for a nightly document classification job that processes ten thousand records at a concurrency of one, and for every development environment. The error is exclusively in promoting it to interactive multi-user traffic.

BETTER

Pinned model, bounded queue, continuous batching

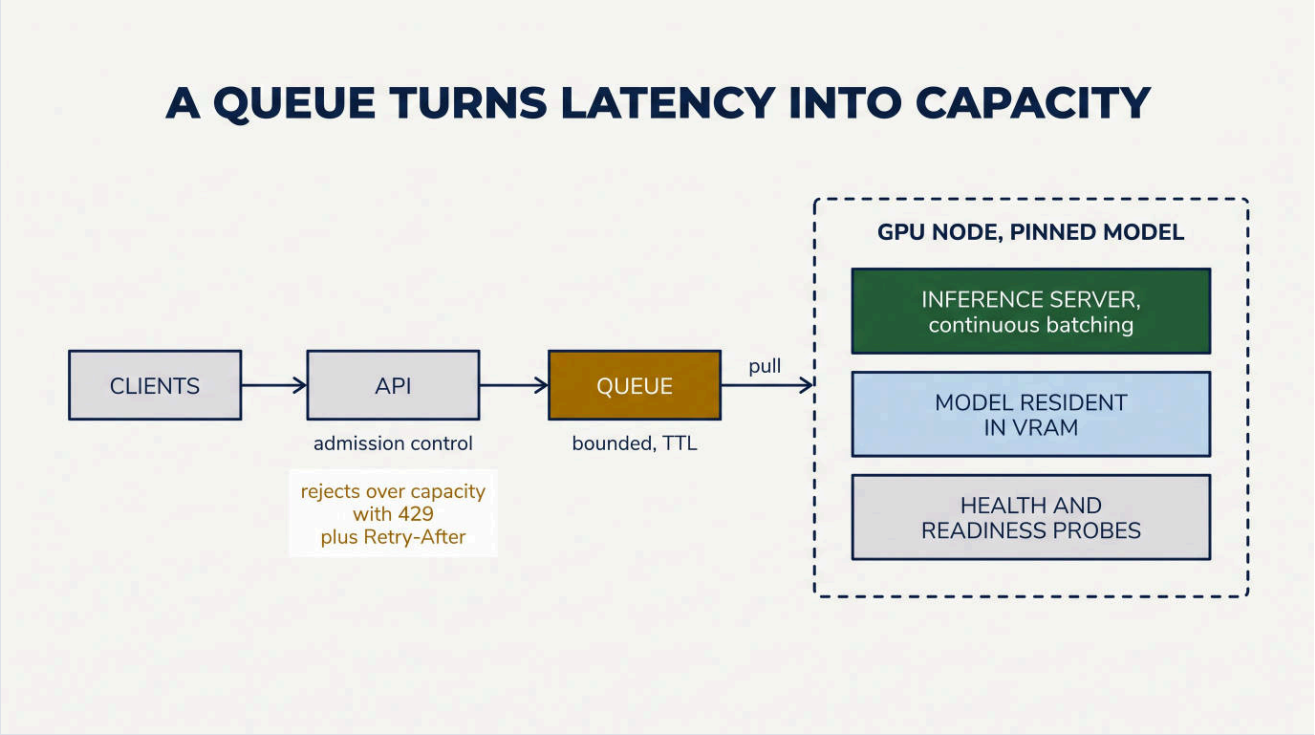

Figure 4.3 The queue converts an unbounded latency problem into a bounded capacity problem, which is a problem you can manage.

Continuous batching is the reason to leave Ollama

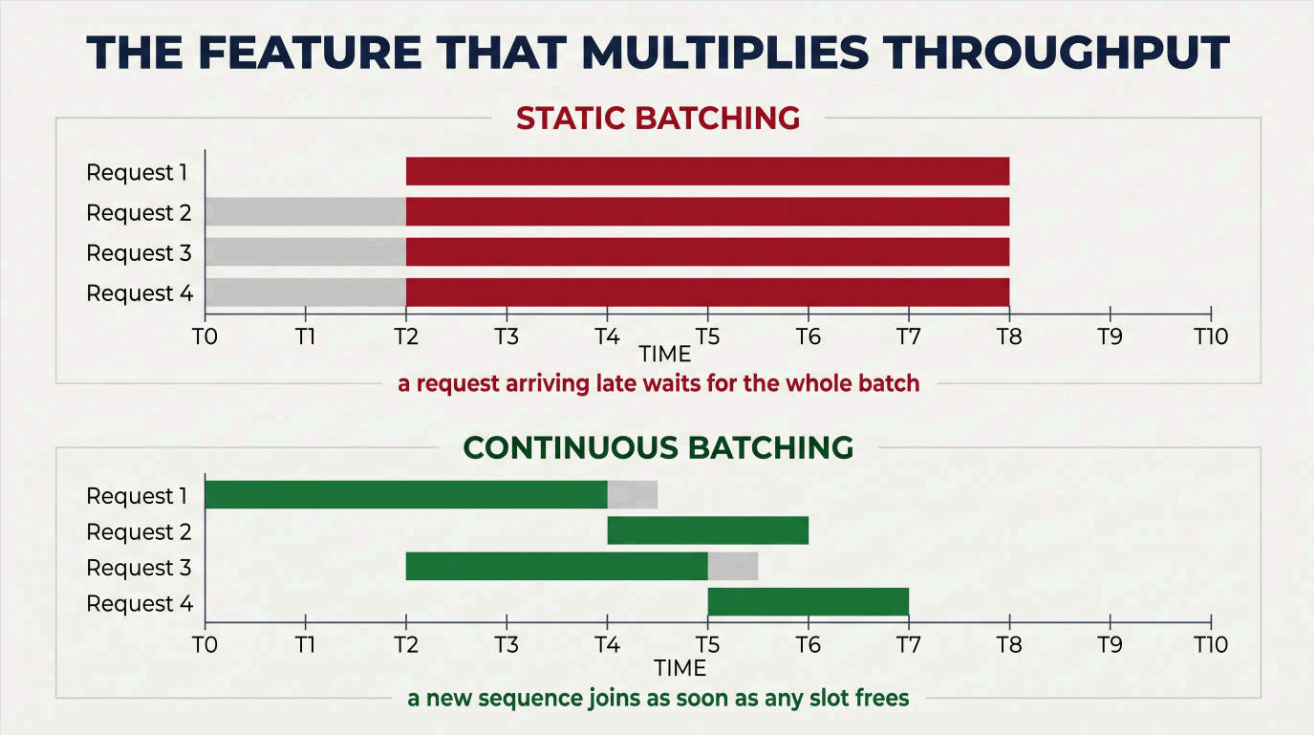

Figure 4.4 Static batching makes every request wait for the slowest in its batch. Continuous batching inserts a new sequence the moment any slot frees.

Naive batching waits for a fixed number of requests, processes them together, then waits again. A request arriving just after a batch starts waits for the entire batch to finish before it is even scheduled. Continuous batching, sometimes called in-flight batching, inserts new sequences into the running batch as soon as any sequence completes.

On realistic mixed traffic this improves throughput by a large multiple, commonly five to twenty times, over sequential processing at the same latency. **This single feature is the main reason to use a purpose-built inference server rather than a convenience wrapper**, and it is why the promotion from Ollama is a change of component rather than a change of settings.

Do not take the multiple on trust, because it is an afternoon to measure. Replay a fixed set of concurrent requests from

your own traces, fifty is enough, against the same quantised model under Ollama and under a batching server on the same hardware, and compare sustained output tokens per second across the run. The ratio you measure is the one that belongs in your capacity plan. On mixed-length interactive traffic it usually lands inside the range above; on uniform batch work it lands lower, because sequences that finish together waste less either way.

Name the destination, because "purpose-built inference server" is a class with concrete members. At the time of writing the default choice is **vLLM**, which introduced paged KV cache management and made continuous batching the standard it now is. **SGLang** competes at the same tier and is often stronger on structured output and heavy prefix reuse. **TensorRT-LLM** extracts the most from NVIDIA hardware at the cost of a compile step and the deepest vendor coupling. And **llama.cpp's own server** covers the small case: the same engine Ollama wraps, without the convenience layer, for single-accelerator and CPU deployments.

The names will age; the properties to look for will not. All of them do continuous batching. All manage KV memory in pages rather than contiguous reservations, which is what lets one long conversation and twenty short ones share a card without fragmentation. And all expose an OpenAI-compatible endpoint, which is the architectural property: the Chapter 3 gateway points at any of them with a base URL change, and no caller learns that the model moved in-house.

The selection heuristic in one sentence: start with vLLM; move to TensorRT-LLM only if you are pinned to NVIDIA's toolchain and chasing the last quartile of throughput; drop to llama.cpp's server when you are serving one small model on modest hardware and the batching multiple has nothing to multiply.

Admission control that actually refuses

```
# src/docassist/serving/admission.py
class AdmissionController:
    """Bounded concurrency plus a bounded wait.

    A request that cannot be served within its own deadline should be refused
    NOW rather than served late. A late answer usually has no value to the
    user who asked for it, and it has consumed capacity that a fresh request
    could have used.
    """

    async def __aenter__(self) -> "AdmissionController":
        # Refuse at the door, before occupying a queue slot.
        if self._queued >= self.max_queued:
            self.stats.rejected_full += 1
            raise AtCapacity(retry_after_s=2, reason="queue full")

        self._queued += 1
        started = time.monotonic()
        try:
            await asyncio.wait_for(self._sem.acquire(),
                                   timeout=self.max_queue_wait_s)
        except asyncio.TimeoutError:
            self.stats.rejected_wait += 1
            raise AtCapacity(retry_after_s=5, reason="queue wait exceeded") from None
        finally:
            self._queued -= 1

        self.stats.wait_samples.append(time.monotonic() - started)
        self.stats.admitted += 1
        return self
```

Three properties of that code are the whole point.

Rejection carries `Retry-After`. A client must be able to distinguish *busy* from *broken*. Without it, clients retry immediately into a saturated system, which is the retry storm of section 3.6 arriving from your own front door.

Two distinct rejection reasons, counted separately. `rejected_full` means demand exceeds your queue; `rejected_wait` means the queue is draining too slowly. They call for different responses, and averaging them into one "rejected" counter destroys that information.

The health method reports queue depth and p95 wait, not CPU:

```
def health(self) -> dict[str, object]:
    """Section 16.4: scale on QUEUE DEPTH and time to first token, not CPU.
    A pool can be fully saturated at ten percent CPU because the work is
    dominated by waiting on the accelerator."""
    return {
        "queued": self._queued,
        "max_concurrent": self.max_concurrent,
        "p95_queue_wait_s": round(self.stats.p95_wait_s, 3),
        "accepting": self._queued < self.max_queued,
    }
```

The concurrency limit is arithmetic, not a guess

```
admission = AdmissionController(max_concurrent=23, max_queued=200)
```

Twenty three is not a round number and it is not an experiment. It is the output of `serving_capacity` from section 2.11: available VRAM, minus weights, minus overhead, divided by KV cache per sequence at your typical context length. Section 2.8 showed the same 8B model at 5 or 23 concurrent sequences depending purely on whether it uses grouped query attention.

Setting this by intuition is how teams either waste half an accelerator or discover the limit through an out-of-memory kill. Compute it, and put the computation in a test so that a context-length change surfaces as a failing assertion rather than an incident.

What this tier still does not solve. One model serves every task, so a difficulty classification and a multi-document synthesis receive identical resources. The accelerator is paid for around the clock regardless of traffic, and interactive products idle for two thirds of the day. And there is no answer to a request that genuinely exceeds the local model's capability other than to return a poor answer with full confidence.

BEST

Tiered local with escalation and sampled verification

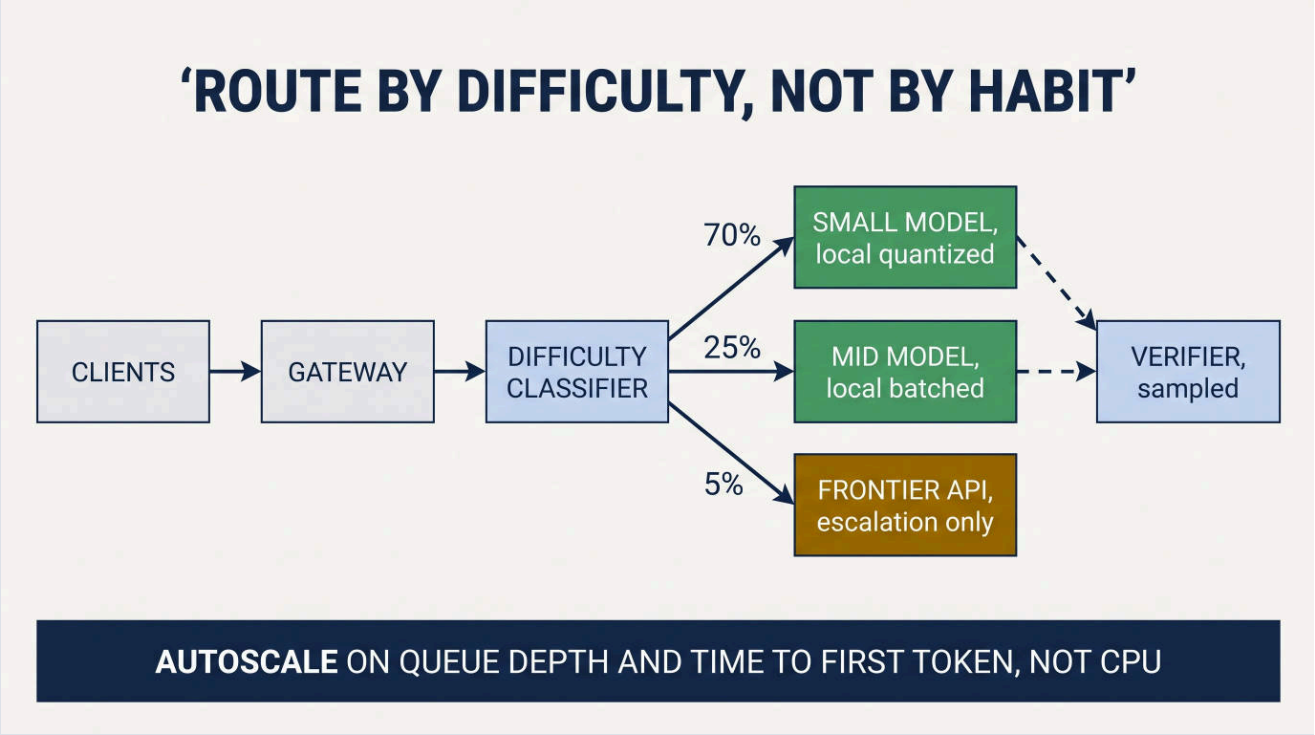

Figure 4.5 Route on predicted difficulty, verify a sample of the cheap tier's work, and let the disagreement rate tune the threshold.

Chapter 3's router selected a tier from the *task type*, using a static table that is free, deterministic and cannot drift. This one selects from predicted *difficulty*, which is none of those things, and is therefore only worth building once the task-type table has stopped being enough.

```
# src/docassist/serving/escalation.py
class EscalationRouter:
    SMALL_CEILING = 0.35
    STANDARD_CEILING = 0.80
    LONG_CONTEXT_OVERRIDE = 30_000

    def _tier_for(self, score: float, task: LocalTask) -> Tier:
        tier = (Tier.SMALL if score < self.SMALL_CEILING
                else Tier.STANDARD if score < self.STANDARD_CEILING
                else Tier.FRONTIER)
        # Hard overrides a classifier should never be trusted to catch,
        # because they depend on the request rather than on its difficulty.
        if (task.evidence_tokens > self.LONG_CONTEXT_OVERRIDE
                or task.requires_citation_precision):
            tier = Tier.FRONTIER
        return tier

    async def run(self, task: LocalTask) -> str:
        score = await self.classifier(task)
        tier = self._tier_for(score, task)
        answer = await self.tiers[tier].generate(task)

        if tier < Tier.FRONTIER and random.random() < self.sample_rate:
            # OFF the critical path. The user never waits for verification.
            t = asyncio.create_task(self._verify(task, answer, round(score, 1)))
            self._background.add(t)
            t.add_done_callback(self._background.discard)

        return answer
```

The verifier is the component that stops this decaying

Without it, this design fails silently over months. Traffic drifts, the classifier's calibration slips, and quality erodes with no signal, *because the cheap tier never reports that it struggled*. A small model does not know it gave a worse answer; it gives a confident one either way.

```
async def _verify(self, task, answer: str, band: float) -> None:
    reference = await self.tiers[Tier.FRONTIER].generate(task)
    agreed = await self.verifier(answer, reference)
    self.stats.record(band, agreed)          # PER SCORE BAND


def bands_below(self, floor: float) -> list[float]:
    """Bands where the cheap tier is no longer trustworthy.

    The response is to RAISE the threshold for those bands, which routes more
    of that traffic upward. That is a tuning action a human takes on evidence,
    not an automatic behaviour, because the alternative interpretation is that
    the VERIFIER itself drifted.
    """
    return sorted(band for band, samples in self.agreements.items()
                  if len(samples) >= 30 and sum(samples) / len(samples) < floor)
```

Recording agreement **per score band** rather than as an average is what makes the signal actionable. An overall agreement rate of 94 percent can hide a band at 0.3 where agreement has fallen to 70 percent, because the bands where the classifier is confident and correct dominate the average. Averaging the bands together destroys exactly the information you need.

PROMOTION TRIGGERS

Good to better, at the first moment two users can plausibly issue a request within the same thirty second window, or when a cold start would be visible to anyone outside the team.

Better to best, when any of these hold: accelerator utilisation is below forty percent while spend is material; a measurable fraction of traffic is failing on local model capability; or *you can demonstrate on your eval set that a small model matches the large one on at least half your task mix*. That last condition is the real gate. It requires an eval set, which is why Chapter 14's material tends to become urgent right about here, and it is also why the task-type router of Chapter 3 should be exhausted first: it captures most of the same saving with none of the calibration risk.

EXERCISE 4.5

Run `serving_capacity` from section 2.11 for the model and hardware you actually have, at your p50 and p95 context lengths. Set `max_concurrent` from the p95 figure, not the p50. Then load-test past the limit and confirm you receive 429 with `Retry-After` rather than a timeout or a restart. If you get a restart, your memory limit is missing from the compose file.

4.6 Configuring and Securing a Hugging Face Account

Three settings, and every one of them is a supply chain control. This is the section where "download a model" stops being a convenience and starts being a dependency decision with the properties of section 1.2.

Fine-grained tokens, scoped read-only

A write-scoped token in a build pipeline is a token that can publish weights under your organisation's name. Scope to specific repositories, read only, and separate tokens per environment exactly as in section 3.1.

Pin by revision hash, never by branch

A model repository's `main` branch can change. Downloading by branch name means your production weights can differ between two deploys of identical code, which is precisely the failure section 1.2 exists to prevent, arriving through a different door.

```
from huggingface_hub import snapshot_download

path = snapshot_download(
    repo_id="google/gemma-2-9b-it",
    revision="8f4f8c0e...",           # a COMMIT HASH, not "main"
    allow_patterns=["*.safetensors", "*.json", "tokenizer*"],
    token=os.environ["HF_TOKEN"],
)
```

The revision hash belongs in the runtime fingerprint from section 1.2, alongside the lockfile hash. When behaviour shifts, "which weights were running" must be answerable from a trace.

A third check belongs next to the revision pin: the licence. Open weights are not uniformly open. Community licences carry use restrictions and user-count thresholds, some weights are released for research use only, and a fine tune inherits the licence of its base. Record the licence in the model registry entry alongside the revision hash, and gate deployment on it the same way you gate on the eval: a model that fails the licence check does not ship, however well it measures. This is an engineering gate rather than legal advice; the point is that the check is written down and runs before the weights reach production, not after.

Prefer safetensors, and mean it

THE PICKLE PROBLEM

Pickle-based checkpoint formats **execute arbitrary code on load**. Downloading a pickle checkpoint from an untrusted repository and loading it is equivalent to running a script from that repository as root on your GPU host, with your cloud credentials in the environment.

Restrict downloads to `*.safetensors` and fail rather than fall back. The `allow_patterns` argument above is doing security work, not bandwidth work.

4.7 Accessing Instruct-Tuned Models

The distinction between a base model and an instruct-tuned model causes more confusion than it should, and the symptom is distinctive enough to diagnose from a single output.

A **base model** continues text. Give it a question and it produces a plausible *continuation*, which frequently means more questions, because that is what follows a question in the training data. It is not broken; it is doing exactly what it was trained to do.

An **instruct-tuned model** has been further trained to follow instructions in a specific chat format. It carries a chat template, and section 5.7 shows that applying the wrong one produces output that is coherent while subtly ignoring the system message.

The rule: use the instruct variant unless you are fine tuning, in which case you usually start from the base. And whichever you use, print the fully rendered prompt at least once, which section 4.9 returns to.

4.8 Installing and Using Hugging Face CLI Tools

```
huggingface-cli login --token "$HF_TOKEN"

# Inspect BEFORE downloading 30 GiB: file list and sizes
huggingface-cli repo info google/gemma-2-9b-it --files-metadata

# Download only what you need, to a shared cache the container mounts
huggingface-cli download google/gemma-2-9b-it \
  --revision 8f4f8c0e --include "*.safetensors" "*.json" "tokenizer*" \
  --local-dir /models/gemma-2-9b-it

# Verify integrity before serving from it
sha256sum /models/gemma-2-9b-it/*.safetensors > /models/gemma-2-9b-it/SHA256SUMS
```

Set `HF_HOME` to a persistent volume shared across containers. Without it, every image rebuild re-downloads tens of gigabytes, which turns a two minute deploy into a twenty minute one and makes rollback slow at exactly the moment you need it fast. This is the same reasoning as the named volume in section 4.2, applied to the build pipeline.

4.9 Model Downloading and Execution from the Hub

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

tok = AutoTokenizer.from_pretrained(MODEL_PATH)
model = AutoModelForCausalLM.from_pretrained(
    MODEL_PATH, torch_dtype=torch.bfloat16, device_map="auto")

messages = [{"role": "user", "content": "Summarise this clause: ..."}]
prompt = tok.apply_chat_template(messages, tokenize=False,
                                 add_generation_prompt=True)
print(prompt)          # PRINT THIS. See section 5.7 for why.

inputs = tok(prompt, return_tensors="pt").to(model.device)
out = model.generate(**inputs, max_new_tokens=300, do_sample=False)
print(tok.decode(out[0][inputs["input_ids"].shape[1]:],
                 skip_special_tokens=True))
```

> TRANSFORMERS IS FOR EXPERIMENTATION, NOT SERVING
>
> The code above runs one sequence at a time with no continuous batching, no paged attention and no request scheduling. Its throughput on a busy accelerator is a small fraction of a purpose-built inference server's, and the gap is measured in multiples rather than percentages.
>
> It is the right tool for evaluating a model, inspecting a chat template, and prototyping. It is the wrong tool for production serving, and using it there is how teams conclude that self hosting is uneconomic when what they measured was the wrong runtime.

4.10 Quantization, GGUF and Hardware Sizing

Quantisation reduces the bytes per parameter. Section 2.3 established where the parameters live, which is why it works so well: roughly two thirds of them are in the feed-forward layers, and those compress cleanly.

Precision	Bytes per param	8B weights	Typical quality effect	Reasonable use
FP16 / BF16	2	~16 GiB	Reference	Training; serving where memory is abundant
INT8 / FP8	1	~8 GiB	Near indistinguishable on most tasks	The usual production default
4-bit (AWQ, GPTQ)	~0.55 with scales	~4.5 GiB	Small but measurable; worst on long chain reasoning and rare vocabulary	High volume, latency sensitive, simpler tasks
3-bit and below	<0.45	~3.5 GiB	Noticeable and unevenly distributed across task types	Rarely justified outside memory-constrained edge

GGUF is the container format used by llama.cpp and therefore by Ollama. Its suffixes encode the scheme: `Q4_K_M` is a four-bit K-quant medium mix, the common default and a reasonable one. `Q5_K_M` costs about twenty percent more memory for a small quality gain. `Q8_0` is effectively lossless and rarely worth the memory over INT8 in a server runtime.

The trap that matters

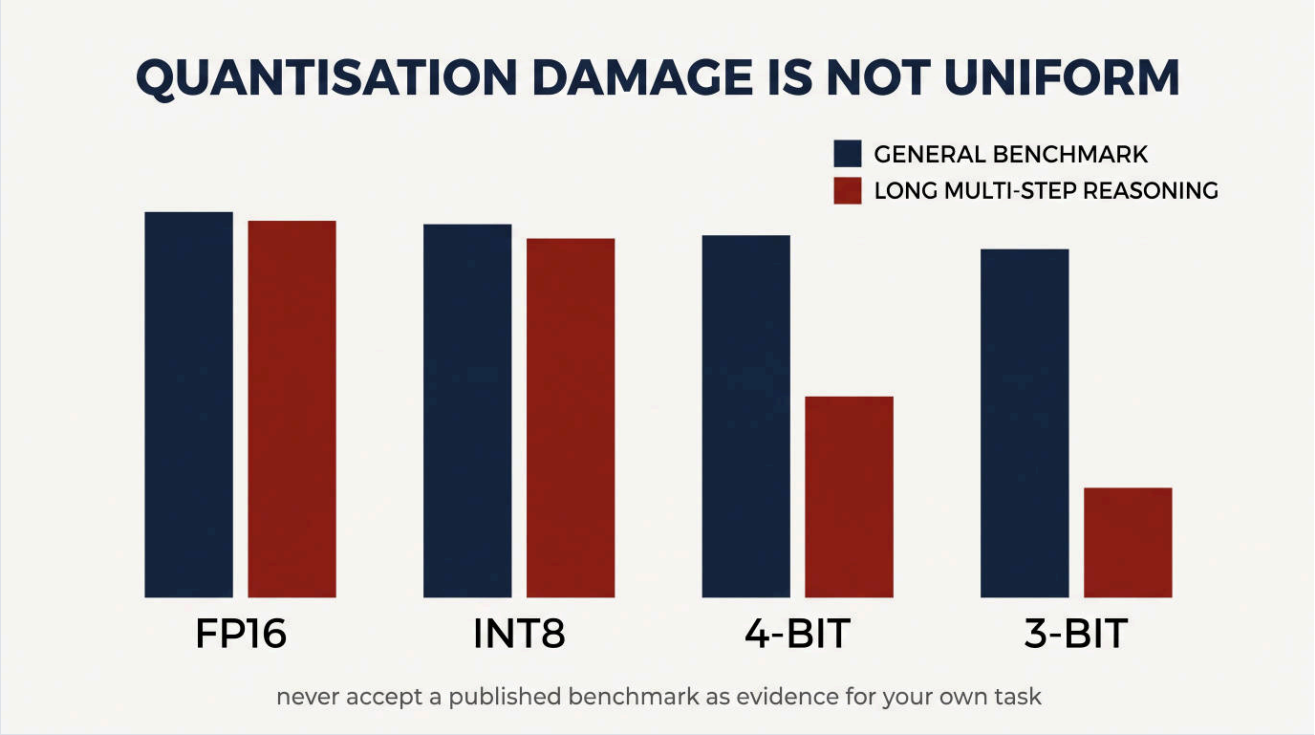

Figure 4.6 Indicative shape. A general benchmark can stay nearly flat while the task you actually run degrades sharply.

Quantisation damage is **not uniform across task types**. A four-bit model can score within a point of its full-precision version on a general benchmark while losing far more on your specific task, particularly where the task involves long multi-step reasoning, exact numeric extraction, or rare domain terms.

Section 2.4 explains part of the mechanism: rare domain vocabulary is represented by fragmented, low-frequency tokens, and low-frequency behaviour is what quantisation degrades first. A general benchmark is dominated by common vocabulary, so it is close to blind to this.

Never accept a published benchmark as evidence that quantisation is safe for your workload. Run your own eval set at each precision. It is an afternoon of work and it regularly changes the decision.

And remember what quantisation does not touch

Section 2.11's finding, restated because it governs the sizing conversation: quantisation shrinks the weights band only. It does nothing to the KV cache. On the repository's numbers, four-bit weights took an 8B model from 23 to 34 concurrent sequences at 8k context, and allowing conversations to reach 32k dropped it to 8. Context length policy moves the number further than precision does.

One further sizing caveat. The strongest open weights at the frontier-replacement tier are now largely mixture-of-experts, where the distinction from section 2.3 applies: weight VRAM follows total parameters while throughput follows active parameters, so the dense arithmetic above is a lower bound story on memory and a conservative one on speed. Size the card for the total; predict tokens per second from the active.

4.11 Decision Guide: Local Weights versus Hosted APIs

Four reasons justify the operational burden. Only one of them is cost, and it is the weakest.

Reason	Why it decides the question
Data residency or confidentiality	A contractual or regulatory obligation that text cannot leave your infrastructure. The strongest reason, because no provider assurance substitutes for it.
Latency floor	A small quantised model on a local accelerator returns first token in tens of milliseconds. A hosted frontier model over the public internet cannot, ever. Decisive for inline completion and for the voice work in Chapter 13.
Behavioural stability	Pinned weights do not change under you. For a regulated workflow where an output change requires revalidation, this converts an unmanageable risk into a scheduled one. It is property five of section 1.6, purchased outright.
Cost at sustained high volume	The weakest reason and the one most often cited first. It is arithmetic, and the arithmetic is below.

The arithmetic, and the variable people omit

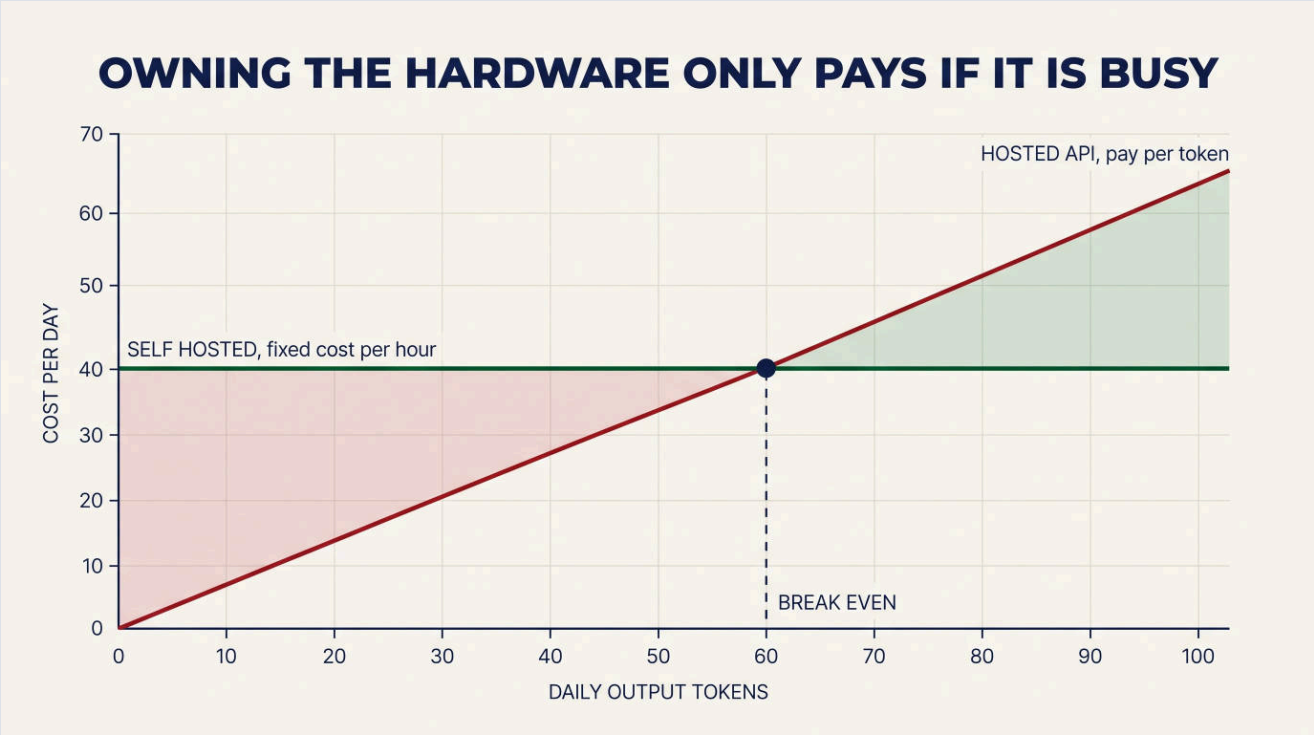

Figure 4.7 Schematic. An accelerator costs the same whether it is busy or idle, so the flat line does not move when your traffic drops.

The trap is comparing an hourly rate against a monthly API bill without computing **utilisation**. The repository makes it explicit:

```
# src/docassist/serving/economics.py
@dataclass(frozen=True)
class SelfHostQuote:
    hourly_usd: float                  # the instance, whether busy or not
    output_tokens_per_s: float         # sustained, at healthy batch sizes
    utilisation: float = 0.35          # THE number people omit

    @property
    def daily_output_tokens(self) -> float:
        return self.output_tokens_per_s * 86_400 * self.utilisation

    @property
    def usd_per_m_output(self) -> float:
        return self.daily_usd / (self.daily_output_tokens / 1_000_000)
```

One accelerator at $1.20 per hour, sustaining 90 output tokens per second. That is $28.80 per day, fixed. Here is the sweep:

```
=== vs $0.50 per M output  (a cheap hosted model) ===
  util 15%   1.17M tok/day  self $24.69/M  api $ 0.58/day vs own $28.80  -> RENT
  util 35%   2.72M tok/day  self $10.58/M  api $ 1.36/day vs own $28.80  -> RENT
  util 60%   4.67M tok/day  self $ 6.17/M  api $ 2.33/day vs own $28.80  -> RENT
  util 85%   6.61M tok/day  self $ 4.36/M  api $ 3.30/day vs own $28.80  -> RENT

=== vs $10.00 per M output  (a frontier model) ===
  util 15%   1.17M tok/day  self $24.69/M  api $11.66/day vs own $28.80  -> RENT
  util 35%   2.72M tok/day  self $10.58/M  api $27.22/day vs own $28.80  -> RENT
  util 60%   4.67M tok/day  self $ 6.17/M  api $46.66/day vs own $28.80  -> OWN
  util 85%   6.61M tok/day  self $ 4.36/M  api $66.10/day vs own $28.80  -> OWN
```

Two findings, and the second is sharper than the received wisdom.

Self hosting a small model to save money loses at every utilisation. Against a cheap hosted model it is not close, even at 85 percent busy: $4.36 per million owned against $0.50 rented. No amount of hardware procurement skill fixes an order of magnitude.

Against a frontier model, utilisation decides the answer, not the hourly rate. The same accelerator against the same API price flips from losing to winning purely on how busy you keep it. At 35 percent it is still losing, and only just: $27.22 rented against $28.80 owned. At 60 percent it wins comfortably.

The rule that follows. Utilisation is a property of *your traffic shape*, not of your hardware choice. A team with spiky, daytime-only, single-region traffic can be at 20 percent and will never win, however good their procurement. Before proposing a fleet, compute your actual duty cycle from a week of traces. If it is below roughly 50 percent and you cannot batch overnight work onto the same hardware, the arithmetic has already answered.

```
def test_against_a_frontier_model_utilisation_decides_the_answer():
    """Same hardware, same API price, opposite conclusions."""
    rows = utilisation_sweep(hourly_usd=1.20, output_tokens_per_s=90,
                             api_usd_per_m_output=10.00)
    by_util = {r["utilisation"]: r["self_hosting_wins"] for r in rows}
    assert by_util[0.15] is False
    assert by_util[0.35] is False        # still losing at a third busy
    assert by_util[0.60] is True         # flips somewhere in between
    assert by_util[0.85] is True
```

That test encodes the finding rather than the behaviour. When someone changes a default next year, the failing assertion tells them which argument they broke.

Comparison

Axis	Good: promoted laptop	Better: pinned and queued	Best: tiered escalation
Concurrency	One, effectively	Computed from KV arithmetic	Computed, elastic per tier
Cold start	30 to 60 s after idle	None; model pinned	None on hot tiers
Behaviour at capacity	OOM kill, restart loop	429 with Retry-After	429, or escalate to hosted
Quality ceiling	Local model's ceiling	Local model's ceiling	Frontier, where needed
Drift detection	None	None	Sampled verification per band
Operational burden	Minimal	Moderate: one stateful fleet	High: two fleets, classifier, verifier

PROMOTION TRIGGER FOR SELF HOSTING AT ALL

Do not build the tiered architecture for cost reasons below roughly ten thousand dollars per month of model spend. Below that the engineering time costs more than the saving, and the complexity buys nothing you can measure. Above it, compute your duty cycle first: the sweep above decides the question faster than any procurement conversation.

What Chapter Four Establishes

What exists now	What later chapters put through it
Runtime assertion on context length	Budget-driven required_ctx from the retrieval pipeline *(6.8)*
Admission control with two rejection reasons	Classes of service and reserved capacity *(8.9)*
Queue depth and p95 wait as health signals	Autoscaling on the right signal *(16.4)*
Difficulty router with sampled verification	Judge calibration and drift detection *(14.3)*
Per-band agreement statistics	Continuous regression monitoring *(14.5)*
Utilisation-aware economics	GPU versus CPU serving and cost modelling *(16.2)*

Chapter checklist

Can you...	Section
Name the three Ollama defaults that are wrong for production, and which two fail silently	4.1
Explain why a service should refuse to start rather than truncate evidence silently	4.1
Say why `/alive` must not depend on the model	4.4
Trace the restart loop, and say why adding memory does not fix it	4.5
Explain continuous batching and why it is the reason to leave a convenience wrapper	4.5
Derive your own `max_concurrent` from three numbers on a model card	4.5, 2.11
Say why recording verifier agreement per score band beats an average	4.5
Explain why `allow_patterns` on a model download is a security control	4.6
Say why a general benchmark is close to blind to quantisation damage on your task	4.10
Compute your duty cycle and say which side of the break-even you are on	4.11

What Chapter 5 builds

Chapters 2 to 4 handled the model layer: what it costs, how to reach it, how to run it. Chapter 5 is the transition into the context layer, and it starts with the object that determines output quality more than model choice does. Prompting techniques with their costs and stopping rules attached, the serialization formats that decide correctness when you self host what you just learned to serve, structured output as the real interface, and then the prompt built three times: an inline string, a versioned registry, and a gated artifact whose quality is a release condition.