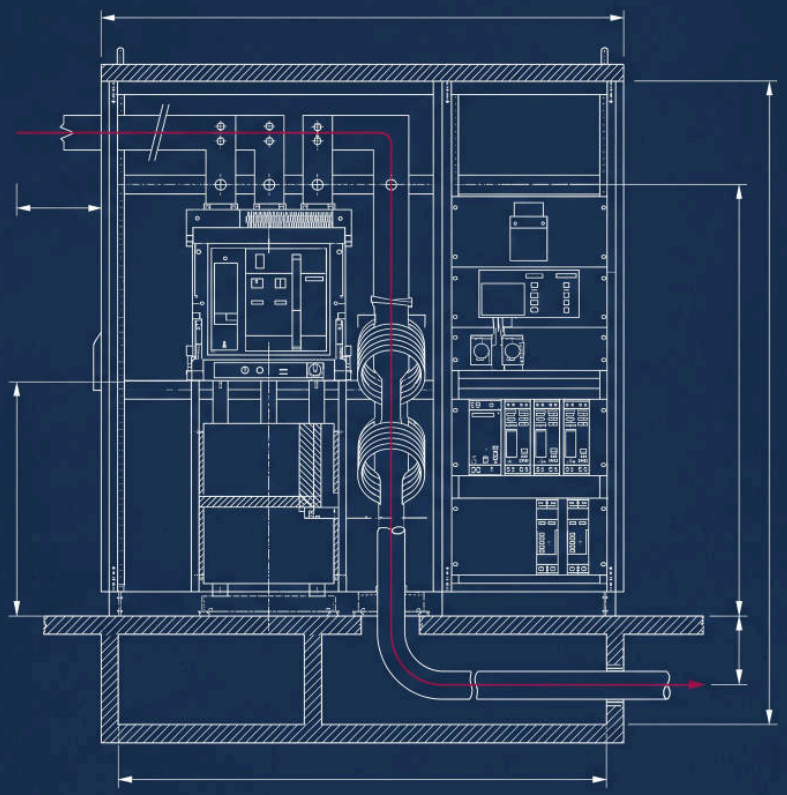

CHAPTER SIXTEEN

Deployment, Scaling and Cost

SECTIONS 16.1 TO 16.10

What changes when the system is not yours to restart at will. Most of the mechanisms already exist, built earlier for other reasons. What changes is that they now answer to someone at three in the morning.

CHAPTER SIXTEEN

Contents

PART IV · PRODUCTION

This is the operational chapter: what changes when the system is not yours to restart at will. Most of the mechanisms already exist, built in earlier chapters for other reasons. What changes is that they now have to answer to someone at three in the morning.

16.1 What Changes in Production

Four things are different, and each invalidates an assumption that held while you were the only user.

Change	What it invalidates
You cannot restart at will	Chapter 8's in-flight work is someone's request. Section 8.5's grace period stops being hygiene and becomes a correctness requirement.
Cost has an owner	A number on an invoice becomes a margin conversation. Section 1.6's cost per successful task is the unit that survives it.
Quality moves without you	Section 14.5. The release process must cover artefacts that are not code.
Failure is a rate	Section 16.9. Nothing errors; a number moves. Conventional alerting has no opinion about this.

16.2 Cost Attribution and Budgets

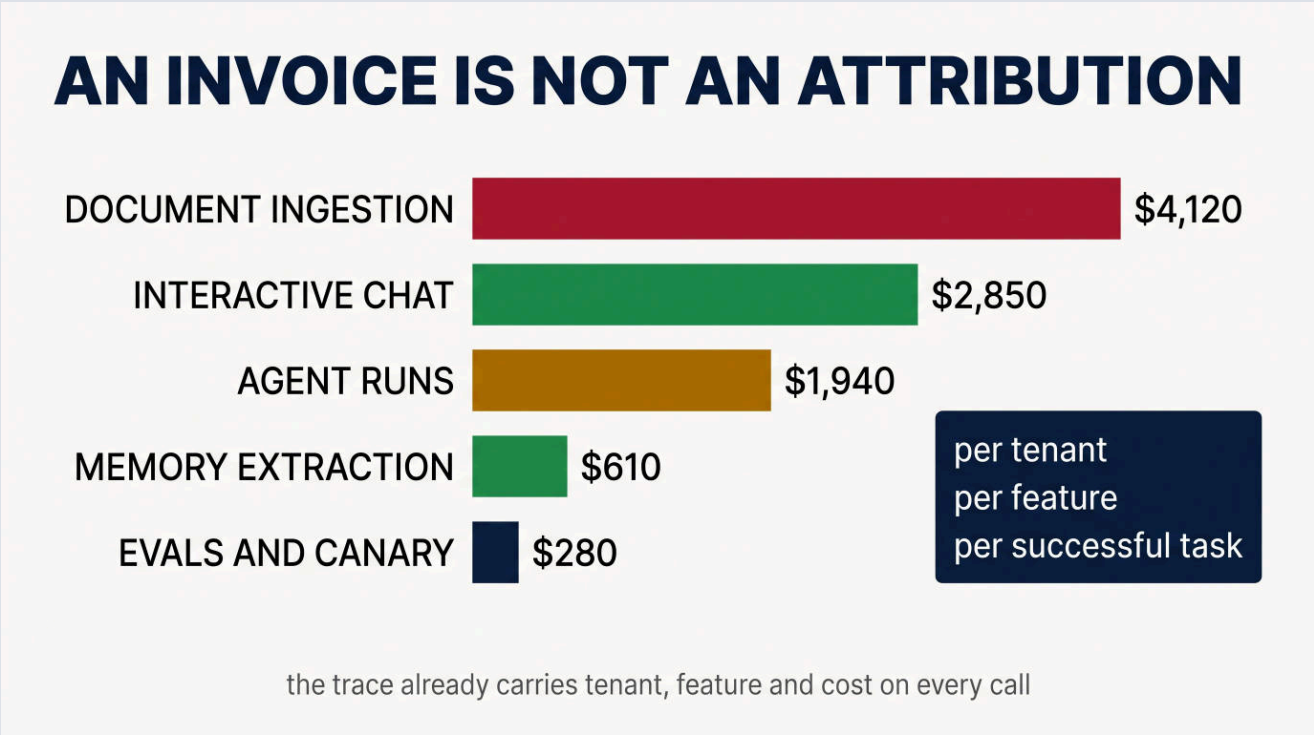

Figure 16.1 The trace already carries tenant, feature and cost on every call. Attribution is a query, not a project.

A provider invoice tells you what you spent and nothing about where it went. Every field needed to answer properly has been on the trace since Chapter 1.

```
# Section 1.6's trace fields, aggregated three ways.
by_tenant  = sum(cost_usd) GROUP BY tenant_id
by_feature = sum(cost_usd) GROUP BY task            # doc_qa, ingest, agent
by_outcome = sum(cost_usd) / count(outcome='ok')    # cost per SUCCESSFUL task
```

Three uses, in increasing order of value.

Finding the surprise. Ingestion is frequently the largest line and is invisible to a team watching chat costs, because it runs on a queue at night.

Pricing. Cost per tenant against revenue per tenant is the question a business asks eventually. Section 2.4's token inflation means it varies by customer corpus, so an average is misleading.

Making the ceiling real. Section 3.6's ledger refuses before the call. Without per-tenant attribution the ledger has no number to enforce, and the budget is a forecast.

The two questions to be able to answer within a minute: which tenant costs the most per successful task, and which feature grew fastest month over month. Both are single queries if the trace carries `tenant_id`, `task`, `cost_usd` and `outcome`, and both are archaeology otherwise.

Provider pricing adds a wrinkle the attribution must record. **Cached input is priced differently from fresh input**: a request whose prompt prefix is already in the provider's cache costs severalfold less than the same request on a cold prefix. Cost per successful task therefore drifts unless the trace records the cache hit rate alongside `cost_usd`, because two identical weeks of traffic can differ materially in cost when a deploy resets the cache. This is a different cache from section 3.6's answer cache: that one avoids the call, this one discounts it.

It also makes prompt layout a deployment concern. Provider caches match on a stable prefix, so the fixed material, the system prompt, the tool schemas, the examples, belongs ahead of anything per-request, and a release that edits the system prompt invalidates the fleet's cache and shows up as a cost regression with no code change. That is one more reason the prompt is a release artefact in section 16.3, and one more thing section 14.6's cost gate is for.

Batch pricing is the other meter to route to deliberately. Providers price asynchronous batch requests at roughly half the interactive rate, which maps exactly onto section 8.9's background class: ingestion, summarisation and evaluation runs have no user waiting and should not pay the interactive price. And the hosted-versus-self-hosted crossover of section 4.11 moves when the hosted side is cache-discounted and batch-discounted, so re-run the sweep with your actual hit rate before concluding that self hosting wins.

16.3 Rollout, Canary and Rollback

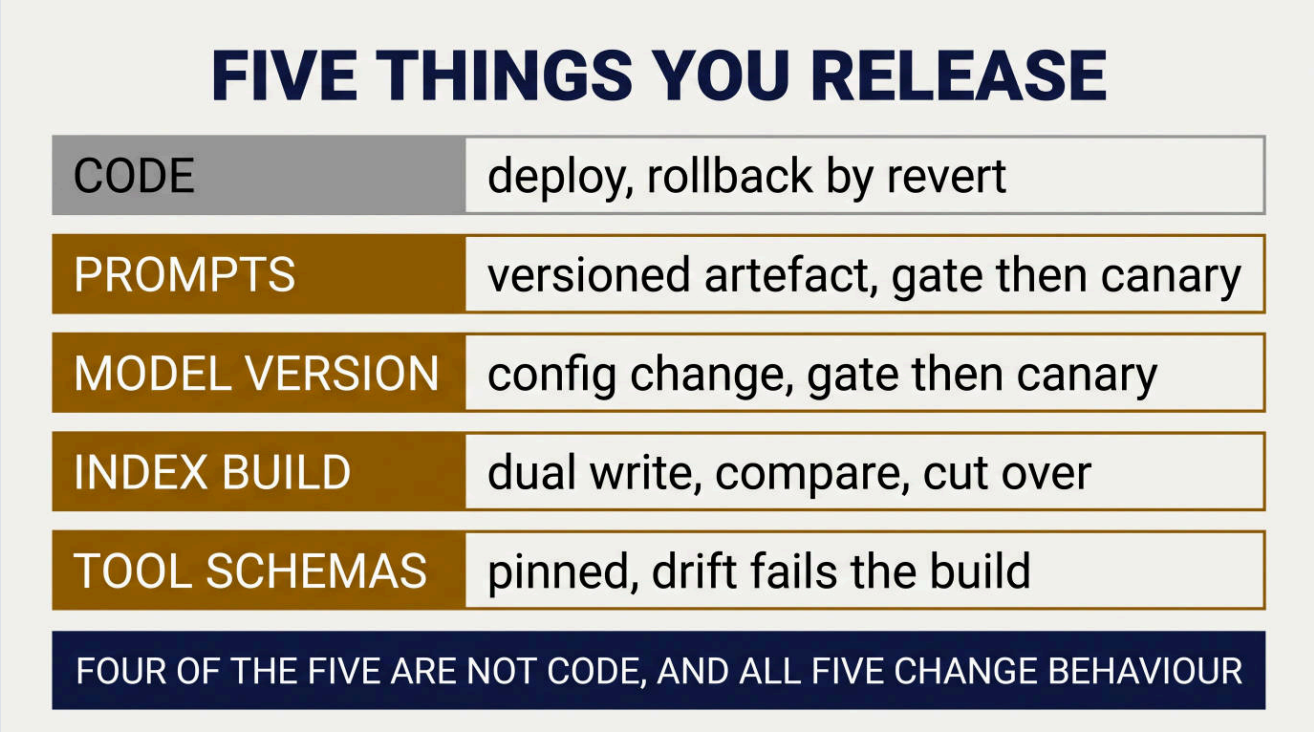

Figure 16.2 Four of the five are not code, and all five change behaviour.

Conventional release process assumes the unit of change is a commit. Here it is one of five things, and only one of them is code.

The implication is that **your deploy pipeline is not your release pipeline.** A prompt change ships without a deploy, a model version changes with a config edit, and an index rebuild changes answers with no artefact in version control unless section 1.2's build id was adopted.

Section 14.7's canary applies to all five. Two additions matter at production scale.

Cohort the canary. Five percent of requests spread across all tenants means every customer sees a mixture. Five percent *of tenants* means a few customers see a consistent experience and the rest see none of it, which is easier to reason about and easier to reverse.

Rollback must not need a deploy. If reverting a prompt requires a release, the argument about whether the regression is real happens before the revert instead of after. Section 5.8's versioned artefact makes rollback a config change, which is the whole point of the artefact.

16.4 Autoscaling on the Right Signals

Section 8.11 established the signal and section 4.5 the concurrency limit. Production adds the question of what to scale when.

```
# Scale on the age of the oldest queued item, per class (8.9, 8.11).
policy = {
    Priority.INTERACTIVE: ScaleOn(max_age_s=20,  out_after_s=30,  in_after_s=300),
    Priority.BATCH:       ScaleOn(max_age_s=600, out_after_s=120, in_after_s=900),
}
```

Three rules, all of which follow from earlier arithmetic.

Age, not depth, and per class. Section 8.11. Thirty seconds is an emergency for interactive and unremarkable for a bulk reindex, so one threshold cannot serve both.

Fast out, slow in. Section 8.11 again: the scale-in window must exceed the longest permitted job, or you terminate work mid-flight and pay for it twice.

Scaling does not fix a saturated accelerator. If the constraint is KV cache per section 2.11, more replicas of a pool that cannot fit more sequences achieves nothing. Know which of the two limits you are against before adding capacity.

Accelerator pools also break the second rule's timing. Scaling out a model pool is minutes, not seconds: the instance boots, tens of gigabytes of weights load, and the cache warms before the replica serves at full rate. Reactive fast-out is therefore unavailable, and the discipline becomes **scale out early, scale in late**: trigger on the queue-age trend rather than the level, and hold capacity longer than the request rate justifies, because giving it back is cheap and getting it back is not.

The mitigations are already built. Pre-pull weights onto standby images, keep a small warm pool ahead of forecast peaks, and let an admission controller shed to section 8.6's queue while capacity arrives; section 4.5's concurrency limit is what tells it when to shed. None of this is new machinery, it is the pieces of Chapters 4 and 8 arranged around a slow start.

16.5 Multi-Region and Data Residency

Section 4.11 listed data residency as the strongest reason to self host, and this is where it stops being a preference.

The complication specific to this domain: **the data does not stay where you put it.** Section 15.4 listed five surfaces where personal data comes to rest, and a naive multi-region deployment replicates only the first.

Surface	The residency question
Documents and vectors	The obvious one, usually handled.
Model inference	Where does the provider actually process? A regional endpoint is a contractual question, not a networking one.
Traces	Contain the document text. Frequently centralised for convenience, which relocates the data.
Memory facts	Section 11.4. Personal by construction.
Caches	Section 3.6's semantic cache holds answers derived from documents.

> THE CHEAPEST CORRECT ANSWER IS USUALLY A SMALLER FOOTPRINT
>
> Full multi-region is expensive and slow to build. Before committing, ask whether the requirement is genuinely "serve from region X" or actually "do not store or process in region Y". The second is often satisfiable by a single-region deployment in the right place, with regional trace retention and a provider endpoint in the same jurisdiction. That is weeks of work rather than quarters, and it satisfies most enterprise questionnaires.

16.6 Provider Incidents and Runbooks

Section 3.6 built the circuit breaker and the degraded path. Production adds the part that is not code: what a person does, in what order, at speed.

Four provider failure shapes, and they need different responses:

Shape	Response
Hard outage	The easy one. The breaker opens, the secondary provider from section 3.3 takes over, and the runbook is mostly "confirm it worked".
Elevated latency	Worse than an outage, per section 3.6's cascade. Tighten deadlines and shed to a lower tier before the queue ages.
Elevated error rate on a subset	Long-context requests failing while short ones succeed. Route by request shape, not by provider health.
Silent quality change	Section 14.5. Nothing fails. This is the one to rehearse, because the response is to pin a dated model version and route to the secondary, and both must already exist.

A runbook that works has three properties: it is **a short ordered list** rather than prose, it **names the dashboard and the command** rather than describing them, and it has been **executed by someone other than its author**. The third is the test; a runbook only its author can follow is documentation.

16.7 Observability and SLOs

Uptime is close to meaningless here. A system returning confident wrong answers at full availability is failing, and an availability SLO scores it perfectly.

Four objectives that mean something, built from what earlier chapters already emit:

Objective	Source
p95 time to first token under target	Section 2.11, and section 13.1 for voice, where it is the only latency number that matters.
Answered rate within a band	Section 6.9. Both directions are failures: falling means retrieval degraded, rising sharply means abstention stopped working.
Sampled quality score above a floor	Section 14.5's production sampling. The only objective that measures the product rather than the plumbing.
Cost per successful task under a ceiling	Section 1.6. An SLO on economics, which most teams never set and every finance team eventually asks for.

Define objectives on service tier, not on success. Section 15.3 gave four tiers, and "the system responded" collapses them. A useful objective reads: ninety-five percent of requests served at FULL tier, ninety-nine percent at FULL or REDUCED. That makes degradation a measured, budgeted state rather than an unreported one.

An SLO without a policy is a dashboard. State the policy: when the window's quality or latency budget is burned, releases freeze except fixes, and the freeze lifts when the budget recovers. That single sentence is what gives section 14.6's gates operational teeth between releases, because it makes the cost of shipping into a burned budget explicit before anyone is arguing about a specific change.

16.8 Capacity Planning

Chapters 2, 4 and 8 supply all three numbers. Planning is the exercise of putting them together before rather than after.

```
# 1. Concurrent sequences per accelerator (2.11), from three model-card fields
serving_capacity(vram_gib=40, shape=LLAMA_8B_GQA, avg_context_tokens=8_000)
#   23

# 2. Throughput ceiling per worker (8.1), which is not a tuning parameter
SyncCeiling(workers=16, generation_seconds=14).requests_per_second
#   1.14

# 3. Where the money crosses over (4.11), decided by utilisation
utilisation_sweep(hourly_usd=1.20, output_tokens_per_s=90,
                  api_usd_per_m_output=10.00)
#   loses at 35% busy, wins at 60%
```

The first number is an illustrative snapshot of a dense model on a 40 GiB card; mixture-of-experts and KV-compression architectures (section 2.3) change the arithmetic's inputs, total versus active parameters and cache per token, but not its structure.

Three planning questions those answer, and the third is the one that changes decisions.

How many accelerators for peak concurrency? Peak concurrent sequences divided by capacity per device, from the KV cache arithmetic rather than from a load test at one context length.

How many workers for peak arrival rate? Section 8.1, remembering that the answer is often absurd and that the

absurdity is the point.

What does a doubling of traffic cost? Not double, if utilisation improves. Section 4.11's sweep means growth can improve unit economics on self-hosted capacity while leaving hosted costs strictly linear. That asymmetry is worth knowing before a growth forecast becomes a budget. Feed the sweep the effective hosted rate after the cache and batch discounts of section 16.2, not the list price, or the crossover lands in the wrong place.

One ceiling is missing from the arithmetic above: the provider's. Section 15.6 rate-limits your tenants, and your provider rate-limits you, in requests and tokens per minute at the account level. Planned peak throughput must fit inside the negotiated limits, and quota headroom belongs on the launch checklist, because raising it is a sales conversation rather than a config change. The secondary provider of section 3.3 earns its keep here too: it is failover and quota diversity at once.

> PLAN AGAINST CONTEXT LENGTH, NOT REQUEST COUNT
>
> Section 2.11's finding stands as the most under-used planning input in this book: quantisation bought fifty percent more concurrency while allowing conversations to grow four times longer gave back four times more than that. A capacity plan that models request growth and assumes constant context length will be wrong in the direction that hurts, because context length grows on its own as products add memory and history.

16.9 The On-Call Experience

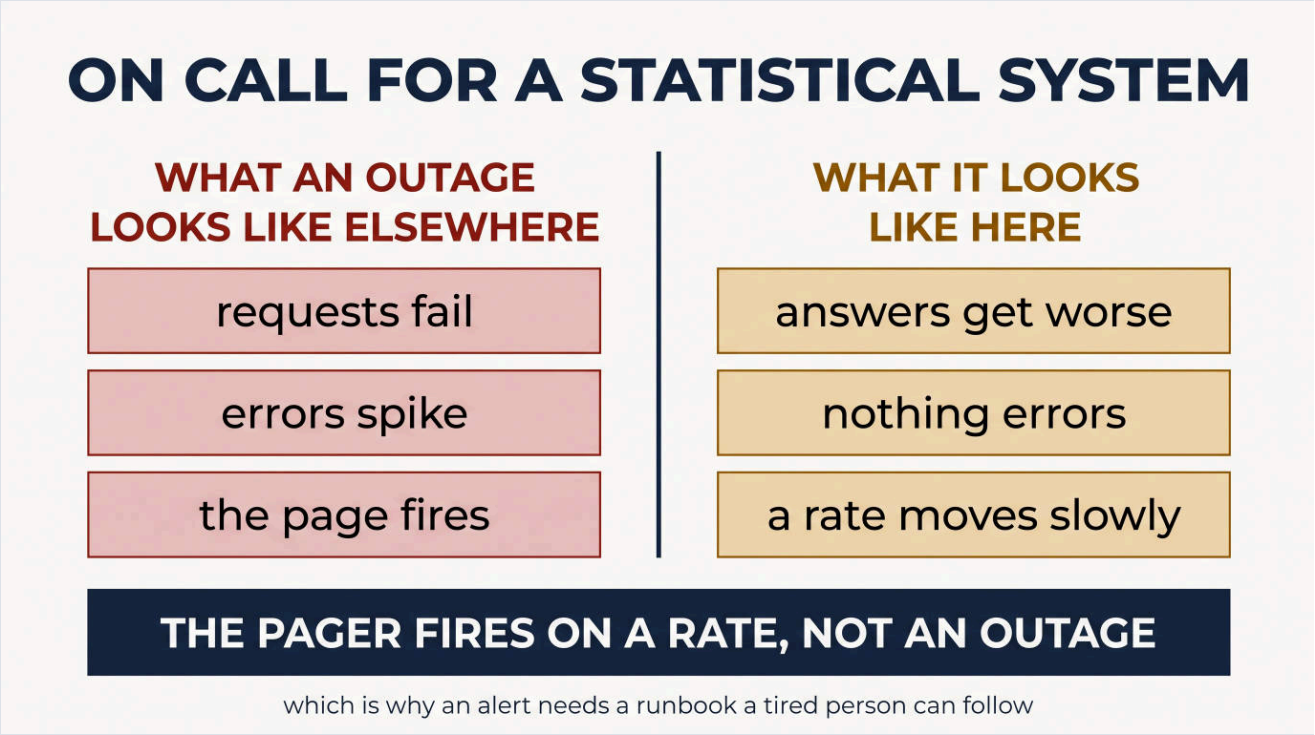

Figure 16.3 The pager fires on a rate, not an outage, which is why the alert needs a runbook a tired person can follow.

Conventional on-call has a comfortable shape: something errors, an alert fires, a person finds the cause and reverts. Here the alert usually says a number moved, nothing errored, and there may be nothing to revert.

Three consequences for how you set it up.

Every alert needs a runbook link, and the runbook needs a first action that is safe. "Answer quality is down four points" is not actionable at three in the morning. "Answer quality is down four points; first action: drop to REDUCED tier with this command; then check whether the model version changed" is.

Alert on the leading proxies, not only the lagging score. Section 14.5's six cost nothing and move first: schema violations, abstention rate, budget truncations, correction rate, denied tool calls, barge-in rate. A sampled quality score is the confirmation, not the alarm.

Separate "worse" from "broken". These want different pages and different response times. A useful split is that *availability and cost* page immediately, while *quality* raises a ticket during business hours unless it crosses a much larger threshold. Waking someone for a two-point quality movement that turns out to be sampling noise is how a team learns to ignore the pager.

A rate alert also needs a sample-size floor before it may fire. Section 14.1's arithmetic applies to the pager exactly as it does to the eval: a rate that moved on twenty requests is noise and on four hundred is signal. Define alert windows in requests rather than minutes, and accept that a low-traffic tenant's alerts arrive later, because the alternative is a pager that fires on arithmetic rather than on the system.

The two questions on-call must be able to answer in five minutes: what version of everything was running when this answer was produced, and what is the safe way to reduce service right now. The first is section 1.2's fingerprint. The second is section 15.3's tier ladder. A system that cannot answer both has an on-call rotation that can only escalate.

16.10 Three Deployment Architectures Compared

GOOD

One service, one region, deploy to release

Everything in one deployable, prompts in the repository, one provider, one region, releases by deploy. Correct for an early product, and it has the real virtue that there is exactly one thing to reason about.

What it leaves unbounded: cost is an invoice; a provider incident is an outage; quality drift is invisible; and a prompt change requires the same ceremony as a schema migration, which means prompts stop being iterated.

BETTER

Separated tiers, versioned artefacts, canaried

API and workers scale independently with classes of service from section 8.9. Prompts, model versions and index builds are versioned artefacts released through section 14.6's gate and section 14.7's canary. Cost is attributed per tenant and feature. A secondary provider exists and has been tested.

What it still does not solve. Objectives are still availability and latency, so quality has no budget. There is no rehearsed response to a silent model change. And residency is a single-region assumption that has not been examined.

BEST

Tiered service objectives and rehearsed operations

Everything above, plus SLOs defined on service tier per section 16.7, autoscaling on queue age per class, capacity planned from the arithmetic rather than from load tests, residency examined across all five surfaces, and runbooks executed by someone other than their author.

Axis	Good: single service	Better: separated and canaried	Best: tiered and rehearsed
Cost	An invoice	Per tenant and feature	Attributed, gated, forecast
Provider incident	Outage	Failover exists	Failover rehearsed, including silent drift
Releasing a prompt	A deploy	Gate then canary	Gate, canary, cohorted, instant rollback
Objectives	Uptime	Uptime and latency	Tier, quality and cost
Capacity	Guessed	Load tested at one shape	Computed, and re-checked on context growth
Build cost	Days	Four to six weeks	Two to three months

PROMOTION TRIGGERS

Good to better, at the first paying customer or the first time a prompt change waits on a deploy window. Both arrive early and both are cheap to fix at that point.

Better to best, when quality has a commercial owner, when an enterprise questionnaire asks about residency, or when the on-call rotation includes people who did not build the system. *The cheapest item on the list is the runbook rehearsal*, and it is the one most likely to matter on the worst day.

What Chapter Sixteen Establishes

What exists now	What the capstone assembles
Cost attributed per tenant and feature	The economics section of the decision log *(17.2)*
Five release artefacts, four not code	What to build first, and in what order *(17.5)*
SLOs on service tier	Reading an unfamiliar system *(17.3)*
Capacity computed from arithmetic	The twenty-minute review *(17.4)*
Rehearsed runbooks	The architect's checklist *(17.7)*

Chapter checklist

Can you...	Section
Name four things that change in production and what each invalidates	16.1
Answer which tenant costs most per successful task, in one query	16.2
Name the five things you release, and which four are not code	16.3
Say why cohorting a canary by tenant beats spreading it by request	16.3
Give the three autoscaling rules and the arithmetic behind each	16.4
List the five surfaces a residency requirement has to reach	16.5
Name four provider failure shapes and the response to each	16.6
Give four SLOs that are not uptime, and say why tiers matter	16.7
Compute accelerators for peak concurrency from a model card	16.8
Say why a capacity plan must model context growth, not just traffic	16.8
State the two questions on-call must answer in five minutes	16.9

What Chapter 17 does

The capstone assembles the whole system on one page and then does something more useful than a summary: it turns sixteen chapters of architecture into a small number of portable habits.

The decision log, which is the artefact that makes an architecture defensible to the person who inherits it. A procedure for reading an unfamiliar AI system quickly, and the twenty-minute review that finds most of its problems. What to build first, which is a short list, and what not to build, which is longer and less popular. And the architect's checklist: eight questions that work on any system in this field, including ones built with tools that do not exist yet.