

The AI Gateway Playbook

Building a Production LLM Gateway with Auth, Quotas, Cost Control, and MCP

The pitch

Every company that gets serious about LLMs eventually hits the same wall. Developer A hardcodes a frontier-model API key into a laptop. Developer B spins up a second key because the first one got rate-limited mid-sprint. The platform team discovers, months later, that the bill has outgrown the team that owns it, that nobody can answer "who is spending what, on which model, for what feature," and that the company's entire AI stack sits on top of a vendor abstraction that was designed before most of today's model providers existed.

This book is the playbook for the system that fixes that: an **in-house LLM gateway** — a thin, boring, auditable layer of infrastructure that sits between your codebase and every model provider, and that treats model access the way any mature platform treats secrets, budget, and access control.

You will learn how to build:

- a **model registry** that automatically syncs models, availability, and pricing across multiple providers, so your codebase talks to stable aliases, not vendor model IDs;
- a **unified API surface** with one wire protocol for chat, streaming, embeddings, and tool-calling, with native pass-through for the long tail;
- a **keyless auth layer** where no static LLM API key is ever issued to a human — credentials live in managed identities and workload brokers, access is granted per role, and on-behalf-of flows attribute every token to a real user;
- a **six-tier RBAC model** adapted for LLM access, with per-role request-rate limits and daily token quotas;
- a **metering and cost pipeline** that accounts every token per model, per tenant, per user, with budgets, soft and hard limits, dashboards, channel alerting, approval workflows for quota increases, and monthly cost reconciliation against provider invoices;
- an **MCP server** that exposes your internal SaaS and office tools as Model Context Protocol tools at 90+ tool scale, secured with OAuth 2.1 + PKCE, envelope-encrypted secrets, PII masking, and a self-service web portal;
- a **RAG-based team assistant** — an orchestrator plus enterprise LLM plus MCP plus retrieval — with a worked end-to-end example (CI/CD failure analysis) and an evaluation harness that keeps it honest;
- and the **operations layer** that makes all of it survivable: audit retention, key rotation without static keys, observability, incident runbooks, a cost-comparison appendix, and deployment and migration checklists.

This is a practitioner's book. Every chapter ends with a pitfalls section — the mistakes that actually happen in the field. Every design comes with a diagram (Mermaid, with ASCII fallback), a config or code sketch, and a decision table where the choice is real.

Who this book is for

- **Senior infrastructure and platform engineers** who own the service mesh, the API gateways, or the developer platform and are being asked to make LLM traffic a first-class citizen.
- **AI platform / ML platform engineers** who need model access to be safe, metered, and attributable before the CFO asks the quarterly question.
- **DevOps and SREs** who will run this thing: you get SLOs, alerting design, runbooks, and a migration checklist.
- **Tech leads and engineering managers** who need to make the build-vs-buy decision and defend it.

You should already be comfortable with: HTTP and gRPC at the protocol level, service meshes or reverse proxies, one major cloud (any major cloud — the book is deliberately cloud-generic), an identity platform with managed identities and keyless service credentials, and basic cost accounting. You do not need to have run a production LLM deployment; if you have, several of the pitfalls sections will read like a warning you were too busy to write down.

What this book is not

- Not a model-ops guide. You will not learn prompt engineering, fine-tuning, or evaluation theory beyond the practical harness the assistant in Chapter 6 needs.
- Not a product manual. Vendor names appear only where the vocabulary is genuinely useful (for example, "the enterprise LLM offering of your cloud," "Entra ID keyless," "M365-family tools," "Atlassian-family tools"). Everything else is described by role, so you can substitute your own stack without re-reading the book.
- Not a case study of any specific company. All numbers are labeled as illustrative. The patterns are general engineering practice; your numbers will be your numbers.

How the book is organized

Chapter	Title	What you get
01	Why companies outgrow managed API gateways for LLM traffic	The cost/lockout/control argument, build-vs-buy decision table, and the TCO framing: why a custom gateway layer can cut managed-premium-style gateway costs by roughly 90%.
02	Reference architecture	The full architecture: model registry with automatic multi-provider sync, unified API surface, routing/limits layer, auth, metering/cost pipeline, audit, observability. Includes a 90-day implementation plan.
03	Keyless auth and the six-tier RBAC model	Never issue static LLM keys: managed identities, OBO flows, six roles, per-role RPS, daily token quotas, approval workflow integration.

Chapter	Title	What you get
04	Quotas, budgets, and cost control	Token/cost accounting per model/tenant/user, budgets and soft/hard limits, real-time dashboards, channel and webhook alerting, quota-increase approvals, monthly reconciliation.
05	MCP at enterprise scale	An MCP server over your internal SaaS and office tools (90+ tools), OAuth 2.1 + PKCE auth, AES-256-GCM envelope encryption for stored secrets, PII masking pipeline, self-service web portal, tool-invocation audit.
06	RAG-based team assistant	Orchestrator + enterprise LLM + MCP + RAG, with a worked CI/CD failure-analysis assistant, guardrails, and an evaluation harness.
07	Operations	Audit logging and retention, key rotation without static keys, observability stack, incident runbooks, cost-comparison appendix, deployment checklist, migration checklist.
08	Production readiness	The full checklist and the common pitfalls, each annotated with why it bites.
99	Index	One-line chapter map for reference.

Read Chapters 1–2 first. Chapters 3–6 are largely independent and can be read in any order, but they share the architecture vocabulary from Chapter 2. Chapter 7 is the operations contract for everything before it; Chapter 8 is what you run before you tell anyone the system is real.

Conventions

- **Code and config:** config snippets are YAML or JSON. Implementation sketches are Go-style pseudocode — real enough to see the shape of the code, annotated where it is deliberately simplified. Where a snippet is pseudocode, a comment says so.
- **Diagrams:** Mermaid code blocks render in most markdown tooling; each critical diagram has an ASCII fallback in a plain code block.
- **Money and volumes:** every number that looks like a price or a volume is *illustrative* and labeled as such. Rebuild the TCO model in Chapter 1 with your own numbers before you use it for a decision.
- **Cloud vocabulary:** "the cloud" = your cloud provider. "The identity platform" = your cloud's identity service (managed identities, keyless credentials, workload identity federation). "The enterprise LLM offering" = the cloud's hosted model service. "The secrets store" = your KMS-backed secret manager.
- **Time markers:** the book assumes a 2026 technology baseline: Model Context Protocol (MCP) is the standard tool protocol, OAuth 2.1 with PKCE is the authentication baseline, reasoning models are a normal cost category, and provider pricing includes cached-token and reasoning-token line items.

A note on confidentiality

Everything in this book is original. The examples are invented and generic: a "large enterprise" with teams of nine, alias names like `standard` and `reasoning`, and config snippets that describe patterns, not any real deployment. If a pattern looks familiar from your own stack, that is the point — the pattern is general, and your specifics stay yours.

One more thing before you start

The single most important sentence in this book is in Chapter 1: **you are not buying a gateway, you are building a cost and control boundary.** The gateway is the easy part. The hard part is the accounting, the access model, and the discipline to make every token attributable. The architecture in Chapter 2 exists to make those three things boring. Everything else in this book is in service of that sentence.

Now let us start with why.

Chapter 1 — Why Companies Outgrow Managed API Gateways for LLM Traffic

"You are not buying a gateway. You are building a cost and control boundary."

This chapter makes the case. It is written for the moment in the room when someone asks: we already have a managed API gateway, why do we need another one for LLM traffic? The honest answer is: because LLM traffic breaks the three assumptions the managed gateway was built on — that a request is a unit of cost, that the vendor abstraction is the product, and that the gateway is a neutral pipe.

1.1 LLM traffic is not API traffic

Managed API gateways were designed in the web-service era. Their cost model, their metering, and their security model all assume **the request is the atomic unit**. An LLM workload breaks that assumption in five concrete ways.

Cost is a function of tokens, not calls. A request to a REST endpoint costs you the same regardless of whether the body is 10 bytes or 100 kilobytes. A request to a model costs you a function of input tokens, output tokens, cached input tokens, reasoning tokens, and — for some modalities — image or audio units. Two requests with identical HTTP shapes can differ by a factor of 10,000 in cost. Any system that meters at the request level is structurally unable to answer the question that matters: *what did this feature, this team, this user cost this month?*

Streams are long-lived. A streaming completion is a Server-Sent Events (SSE) connection that can stay open for minutes. Managed gateways historically bill and buffer by request; long-lived SSE through a managed tier adds per-connection overhead, and in many managed tiers the connection sits behind a shared multi-tenant edge that is not built for 300-second writes. You will feel this as tail latency and, on some platforms, as extra egress or premium-tier surcharge.

Retries are expensive and non-idempotent. A REST call you retry costs the same as the original. A model call you retry *charges you again* — and a partially-streamed response that you discard has already billed most of its output tokens. Retries, failover, and provider outages are therefore *cost events*, not just availability events. Your gateway must account for them, and only a system that sees both attempts and the provider's usage object can do that.

Prompts are payloads, and payloads are data. A model request routinely carries your source code, customer data, or internal documents in the request body. A managed gateway that logs request bodies, samples them for "insights," or routes them through a shared edge is, in many compliance postures, simply not allowed to touch that traffic at all. You need to know, per stream, exactly which component saw which bytes.

The model is a moving target. In any 12-month window, model IDs change, models get deprecated, pricing changes, and a "same" model on a different provider has different limits, different tool-calling

behavior, and different cache semantics. A gateway that treats the model as a URL path on a backend can absorb none of that churn; the churn lands on your developers.

None of these five properties is exotic. They are the normal operating conditions of an organization that has passed the demo stage and started paying a real bill.

1.2 What a managed API gateway was designed for

To be fair to the managed product: it is excellent at what it was designed for.

- Per-request authentication (API keys, JWT validation at the edge)
- Rate limiting and quotas at the request level
- WAF, IP filtering, geo rules, certificate termination
- Basic usage metering — *count of requests*, sometimes bytes
- Blue/green and canary at the route level
- Managed certificates, managed scaling, no server fleet to run

For a REST estate — mobile backends, partner APIs, internal service-to-service — that is a great tool. The failure mode is not that the tool is bad; it is that the LLM workload is *outside the tool's design envelope*, and the managed platform's economics punish you for that:

- **Per-request billing** on exactly the traffic that is cheapest to proxy and most expensive to meter properly.
- **Egress and premium-tier surcharges** on exactly the long-lived, large-payload traffic that LLM workloads are.
- **No token-level metering**, so the feature the gateway exists to provide (usage-based control) is the one it cannot give you.
- **Vendor model abstractions** that predate, or lag, the model market, and that make switching a rewrite.

1.3 The cost problem

Teams commonly see gateway-layer costs drop roughly 90% versus managed-premium-style API gateways when LLM traffic moves to a thin custom layer. Here is why that number is realistic, and why it is not a rounding error.

A managed premium-style gateway bills you for a *platform*: multi-tenant edge, shared VNet integration, per-request fees, per-GB egress, premium-tier instance minimums, and — critically — a floor you pay whether your traffic is 1,000 requests or 1,000,000. The fee is roughly flat per request at the margin, but the platform has minimums: you are paying for the edge, the console, the certificate fleet, and the support tier even in months when the traffic is small.

A custom gateway layer bills you for *compute*: a few stateless proxy instances behind a load balancer, a Redis-sized state store, and a metrics sink. The proxy does one job — authn, routing, metering, streaming

— and does it in a path that is often *faster* than the managed tier, because there is no multi-tenant edge in front of it. The per-request marginal cost of a thin proxy is a small fraction of the managed per-request fee, and there is no platform floor: at zero traffic you pay roughly zero.

Illustrative monthly comparison, all numbers invented to show shape, not to quote anyone's list price:

Cost component	Managed APIM premium-style tier	Custom thin gateway layer
Platform / instance minimum	\$450 /mo (tier minimum, illustrative)	~\$0 (compute only)
Per-request fee (20M req/mo)	\$40 (illustrative \$2 / 1M)	\$0 (compute only)
Egress on large SSE payloads (800 GB)	\$96 (illustrative \$0.12 / GB)	Included in VPC / trivial
Metering, token accounting	\$0 provided — you build it yourself anyway	Built in (Chapter 4)
Streaming tail-latency tax	Shared-edge behavior; often a premium surcharge	None — direct path to provider
Engineering to add token metering	N/A — you must build it out-of-band regardless	Absorbed into gateway work
Illustrative total (gateway layer only)	~\$550 /mo + metering you build anyway	~\$50–150 /mo compute

Two honest caveats, because this is a decision you will defend:

1. The 90% number is about the *gateway layer's* cost, not your LLM spend. Model tokens remain by far the largest line on the AI bill, and this book's real value is in *controlling* that line (Chapters 4–6), not in shrinking it.
2. "Custom" costs engineering time. The breakeven is real: you are trading a managed bill for a small owned component. Chapter 1.6 sizes when that trade is worth it, and Chapter 7's deployment checklist is the price of admission.

The second cost problem is subtler and, in practice, more expensive: **unmetered static keys**. When the managed gateway sits in front of model providers, teams grab provider API keys directly — because the managed gateway gives no per-user attribution, no per-feature budgets, and no reason to route through anything. The bill arrives as one giant number with no decomposition. Chapter 4 is the fix; Chapter 1 is where the wound appears.

1.4 The lockout problem

The vendor abstraction sold to you is not the abstraction you need.

Managed gateways and third-party "AI gateways" of the managed variety typically offer *model routing*: pick from a list of supported providers, get a unified request shape. That product was built on assumptions that are now stale:

- **Model IDs live in the URL or in a vendor-specific field.** When the underlying provider renames a model — and every major provider has renamed models within a 12-month window — the abstraction

has to be re-shipped by the vendor. Your codebase, which was pointed at the vendor's model name, changes with them.

- **Feature coverage is the vendor's schedule, not yours.** Reasoning tokens, prompt-caching, tool-calling variants, batch APIs, fine-tuning endpoints — each provider ships these on its own clock, and the managed abstraction lags by months or omits the long tail entirely. You end up calling the vendor abstraction for the common case and raw provider SDKs for everything else — which means you have two code paths, and the abstraction protects neither.
- **The data path is opaque.** You cannot always tell which provider actually served a request, whether a retry hit a second provider, or what the provider's final usage was. For cost reconciliation (Chapter 4) and for audit (Chapter 3), opaque data path is disqualifying, not inconvenient.
- **Switching is a rewrite, not a config change.** If the abstraction is in the vendor's service, your portability is the vendor's changelog.

The architectural fix is small and it is in Chapter 2: your codebase points at **aliases you own** (`standard` , `reasoning` , `embeddings-1`), the registry maps aliases to provider model IDs, and the mapping is *your* configuration, versioned and reviewed. When a provider renames a model, you change one line in a registry, not one line in forty services. That is the difference between an abstraction you own and an abstraction you rent.

1.5 The control problem

Control has three faces: compliance, attribution, and blast radius.

Compliance. A managed gateway is a data processor you cannot fully inspect. Which components log request bodies? Where do logs replicate? Can you scope the gateway's own credentials? For LLM traffic — which is request bodies, i.e., your data — many compliance postures require that the path be inside your boundary, with your keys, your audit, and your retention rules. A custom layer gives you a component you can put in your VPC, feed to your SIEM, and delete from the architecture if the vendor changes its privacy terms.

Attribution. The question "who used what model, for what feature, and what did it cost" has four axes — user, team/tenant, feature, model — and a managed gateway gives you at most one (the API key that identified the calling service). With static keys shared across a team, even that axis is false. The control fix is structural, not procedural: short-lived identity tokens that carry user, team, and feature scopes, and metering keyed on all four. That is Chapters 3 and 4.

Blast radius. A static provider API key with no scope, no expiry, and no per-user binding is, in security terms, a root credential for your AI spend. If it leaks, the attacker does not read your data — they *burn your budget* at frontier-model prices, and the incident is discovered by the finance team, not the security team. The keyless design in Chapter 3 exists to make that class of incident structurally impossible: no static key exists to leak.

1.6 When to build, when to buy

Do not build this system because it is cool. The managed options are fine for a specific band of situations.

Buy / stay managed when:

- Your LLM spend is small and stable (roughly: under a few thousand dollars per month of model tokens, with no plan to grow).
- You have one model, one provider, and one team.
- Your compliance posture does not restrict the data path.
- You have no capacity to own a small always-on component and its runbooks.

Build when any of these is true:

- Model spend is a *line item* in the budget, not a rounding error.
- More than one team or product consumes models, and "who spent what" is an open question.
- You use more than one provider, or you cannot commit to one.
- You need per-user attribution (compliance, fairness, abuse detection).
- Your compliance posture restricts or forbids the managed data path.
- You are serious about MCP and agentic tool access (Chapter 5) — managed gateways generally do not meter or gate tool invocations.

The build is small. The reference implementation in this book is a thin stateless proxy (one Go service or two), a model registry (a table plus a sync job), a metering pipeline (a stream plus rollups), and an identity integration. A team of two to four engineers, working alongside their existing jobs, can land a v1 in 90 days — Chapter 2 has the week-by-week plan. The alternative is not "managed forever"; it is "managed plus a homegrown metering bolt-on," which costs more and still does not fix the lockout or the static-key problems.

1.7 What you give up by building

Honesty, because the decision will be challenged:

- **You own availability.** The managed tier's uptime is contractual; yours is a pager. Chapter 7 gives you SLOs and runbooks, and the design deliberately keeps the gateway *degradable* — if it goes down, direct-provider calls with scoped keys can continue at reduced control (see 7.2), not full outage.
- **You own the security surface.** You are now a credential broker. That is a privilege, and a responsibility: key hierarchy, rotation, break-glass. All of it is in Chapters 3 and 7.
- **You own the boring parts.** Retention, reconciliation, dashboards, approvals. This book is mostly the boring parts, because the boring parts are where the value is.
- **You will be asked to explain the 90% number to finance.** Bring the table from 1.3 with your own numbers in it, plus the caveat that it is the gateway layer, not the token spend.

1.8 The frame to carry forward

Three sentences, one per problem:

- **Cost:** LLM traffic is metered in tokens, and the only component that sees tokens is the one in front of the model calls — so the metering lives there, and the thin proxy that does the metering costs a fraction of the managed tier that pretends it does not.
- **Lockout:** the abstraction you own is the alias registry; the model ID is a detail of the registry, not of your codebase.
- **Control:** no static keys, per-user attribution, in-boundary data path, and an audit trail that names the human, the team, the feature, and the model for every dollar.

The next chapter is the architecture that makes those three sentences true.

1.9 Pitfalls

- **The "just put it in the managed gateway" meeting.** The counter is structural, not political: token metering, per-user attribution, and data-path control are not config options on the managed product. Ask to see where per-token usage is metered; the answer will be "we don't."
- **Confusing the 90% with free.** The savings are in the gateway layer's bill. Your token bill is unchanged by which proxy sits in front of it, and it is where Chapters 4–6 go to work.
- **Building before the pain is real.** If you have one team and one model and a small bill, a managed tier or even a provider SDK in one service is acceptable. The build is justified by multi-team attribution, multi-provider reality, or compliance — not by architecture taste.
- **The lockout is the vendor's model ID in your URLs.** Audit your codebase for raw provider model names before you build; the registry (2.3) is the retirement of that habit, and you cannot migrate what you have not inventoried.
- **Treating the managed gateway as a provider abstraction it is not.** Many managed tiers are *edge* products that route to your backends; they do not proxy to model providers at all. If your plan was "route LLM calls through the existing APIM," check that it can egress to the provider APIs, stream SSE, and meter tokens. It may be able to do the first and fail the other two.

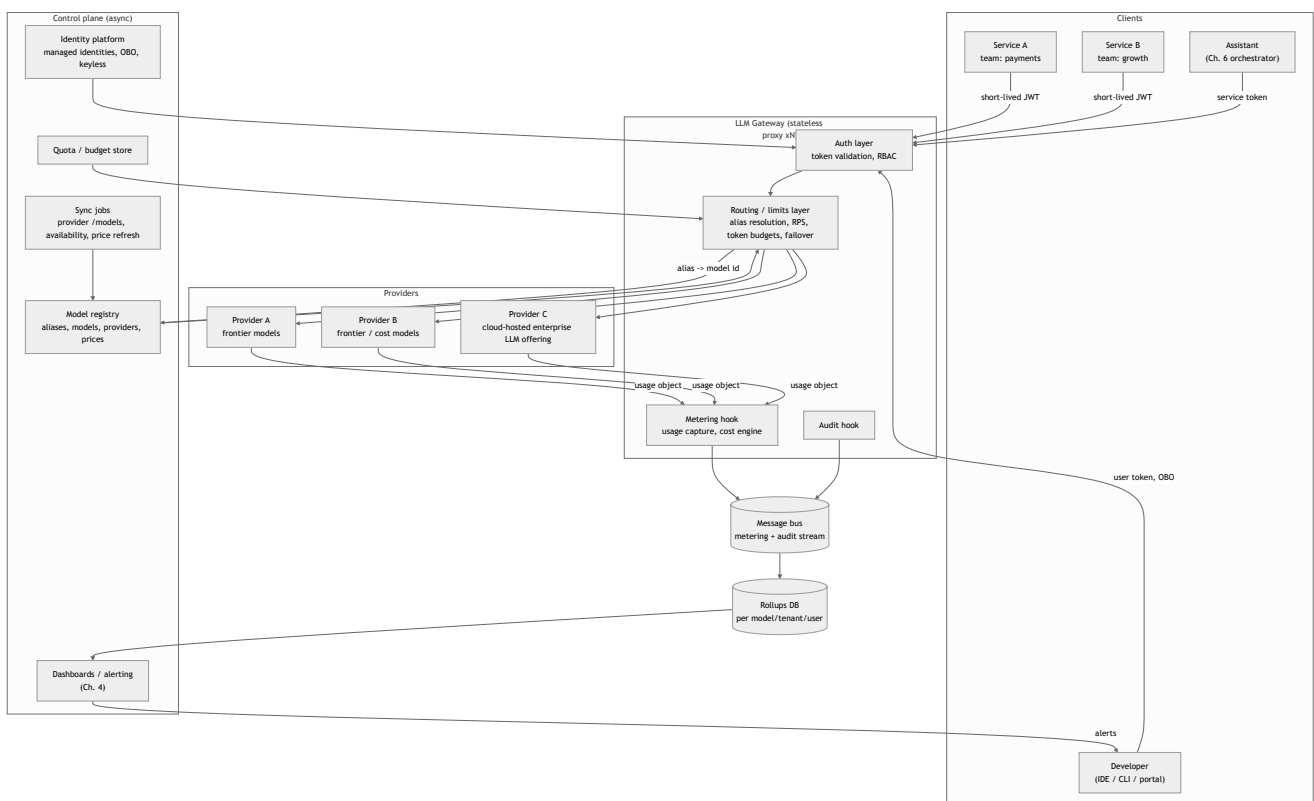
Chapter 2 — Reference Architecture

This chapter is the map. Every later chapter is a zoom-in on one of these boxes. Read it for the shape; go back to it whenever a later chapter says "the metering pipeline" without repeating what that means.

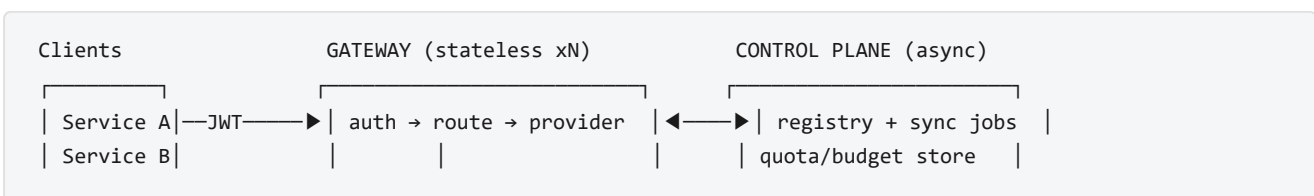
Design stance, stated up front because it is the difference between a gateway and a platform:

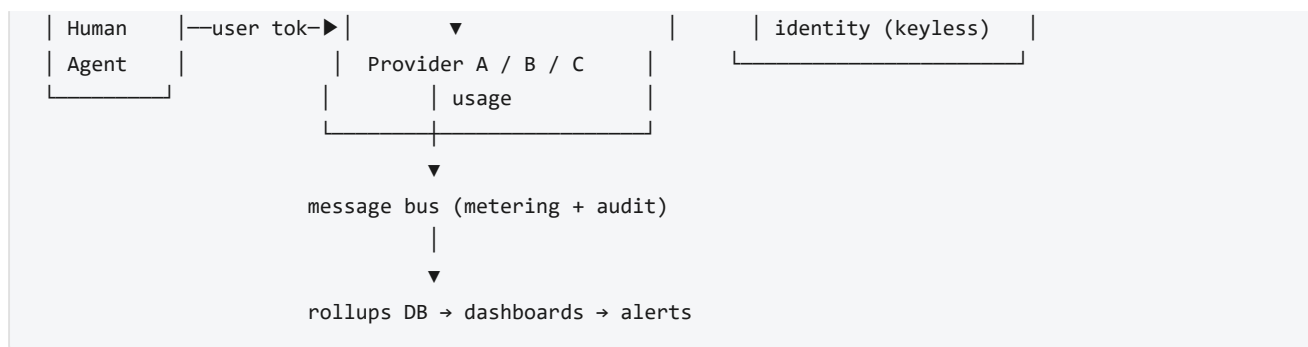
1. **One control plane, one data plane.** Everything a human changes lives in the control plane (registry, roles, budgets, prices). The data plane (the proxy) is stateless and only *reads* control-plane state.
2. **Everything is metered.** If a request does not produce a metering event, it did not happen. The metering event is as real as the HTTP response.
3. **No secrets in requests.** The only credentials on the wire are short-lived identity tokens. Provider credentials never leave the gateway process's secret view (Chapter 3).
4. **Boring technology.** One language, one message format, one stream. The gateway's job is to be the least interesting component in the architecture.

2.1 The big picture



ASCII fallback:





The load path is: clients → auth → routing → provider → (stream back) → metering → bus → rollups. The control plane is never in the hot path except as reads from a cache; it writes state, and the gateway picks up changes on a short TTL (1–5 seconds is fine).

2.2 Component inventory

Component	Runs as	State	Failure mode
Proxy (gateway)	Stateless HTTP/SSE service, N instances	None (caches registry, quotas)	Instances independent; LB spreads
Model registry	Table in a small Postgres/SQL DB	Aliases, models, providers, prices	Read cache; stale read is acceptable, missing registry is not (fail closed on unknown alias)
Sync jobs	Scheduled workers	None	Last-good snapshot retained; stale price is flagged, not fatal
Quota/budget store	Redis (counters) + SQL (policies)	Counters, budgets, limits	Counters best-effort under high load (over-admit, alert); policies fail closed
Auth layer	In-process in proxy (library)	JWKS cache	Token verification fails closed; JWKS refresh is background
Metering hook	In-process in proxy	None	Bus write is queued with bounded retry; metering loss is an incident (see 2.11)
Message bus	Your existing stream (Kafka/Event Hubs/etc.)	Durable	Standard durability
Rollups DB	SQL (the same DB as registry is fine at v1)	Usage facts, rollups	Lag is OK; loss is not (recompute from bus)
Dashboards/alerts	Your observability stack	Queries	Standard
Web portal (Ch. 5)	Small app in front of control plane	UI state only	Non-critical path

Note the deliberate absences: no separate "AI platform" microservice forest, no ML orchestration framework in the hot path, no second database. At v1 this is one deployable plus scheduled jobs. You can split components later; the interfaces below are the seams.

2.3 Model registry with automatic multi-provider sync

This is the lockout fix from Chapter 1, made concrete.

Two tables, one job.

`providers` — the external reality:

```
# providers.yaml (imported into providers table; YAML shown for review-ability)
- id: provider-a
  endpoint: https://api.provider-a.example/v1
  auth: managed-identity      # resolved by gateway, never in config values
  capabilities: [chat, streaming, tools, reasoning, cache]
  status: active
- id: provider-c-enterprise  # "the enterprise LLM offering of your cloud"
  endpoint: https://llm.enterprise.example/openai/v1
  auth: managed-identity
  capabilities: [chat, streaming, tools, reasoning, embeddings, batch]
  status: active
```

`models` — the catalog. Each row is one (provider, model_id) with its *observed* state:

```
- alias: standard          # what codebases may name
  provider: provider-c-enterprise
  model_id: enterprise-gpt-4o-class-2026-01  # vendor detail, hidden from callers
  capabilities: [chat, streaming, tools, reasoning, cache]
  context_window: 128000
  price_per_mtok: { input: 2.50, output: 10.00, cached_input: 0.25 } # USD, refreshed
  status: active           # active | deprecated | unavailable
  observed:
    last_sync: 2026-01-15T04:00:00Z
    availability: ok        # ok | degraded | down (from probes)
    deprecation_notice: null # or: "2026-04-30: provider sunset"
```

`aliases` — what your organization owns:

```
- alias: standard
  policy: { min_capability: [chat, tools], max_cost_rank: 2, prefer: provider-c-enterprise }
  binding: standard->provider-c-enterprise/enterprise-gpt-4o-class-2026-01
  canary: { alias: standard-v2, traffic: 5% }
- alias: reasoning
  binding: reasoning->provider-a/frontier-reasoning-2026
- alias: embeddings-1
  binding: embeddings-1->provider-c-enterprise/embed-3-class
```

The sync job is three small loops on a schedule (default: every 15 minutes for availability, daily for price, hourly for deprecation notices; tune to your provider's change velocity):

1. **Catalog sync:** call each provider's model listing endpoint (where one exists) and diff against `models`. New model → row with `status: unreviewed` (visible in portal, not routable until a human approves). Removed model → keep the row, mark `status: gone`, alert. *Never auto-delete a binding.*

2. **Availability probe:** for each `active` binding, fire a tiny, cheap, marked probe request (a one-token completion on the cheapest active model of that provider, tagged `probe: true` in metering so it is visible and excludable from business cost). Track failure rate; flip `availability` and page when a provider that carries a *primary* alias degrades.
3. **Price refresh:** pull the provider's published price list (API or parsed page, per provider) into `price_per_mtok`; if the published price moved more than a threshold (say $\pm 10\%$) vs. the stored value, alert — because your *observed* cost (from the usage object, Chapter 4) and your *booked* price (what you budget) can drift, and that drift is a reconciliation finding, not a bug.
4. **Deprecation watch:** where the provider exposes deprecation metadata (or its changelog is machine-readable), record `deprecation_notice` on affected models and open a ticket in your tracking system. The ticket's fix is a registry change, not a code change — which is the point of the registry.

Why automatic sync and not manual cataloging: the catalog's value is that it is *true*. A hand-maintained catalog rots within a quarter and people stop trusting it, at which point the alias layer is just a slower lie. Automatic sync plus human approval for *bindings* (what callers can name) is the split: facts sync, policy is reviewed.

2.4 Unified API surface

One wire protocol for the common case, pass-through for the long tail.

Canonical endpoints (OpenAI-compatible shapes, deliberately):

```
POST /v1/chat/completions      # streaming (SSE) and non-streaming
POST /v1/embeddings
POST /v1/responses             # tool-calling / agentic loop shape (2026-era)
GET  /v1/models                # your aliases, not provider catalog
GET  /v1/usage/me              # own usage, per role (Ch. 4)
```

- The request body is the common shape: `model` **must** be one of your aliases; raw provider model IDs are rejected (that is the lockout firewall, enforced in one place).
- The gateway appends nothing to the prompt; it adds *headers* on the wire to providers: provider-specific model ID, your provider credential (never visible to callers), and a correlation ID.
- **Usage object:** every response — including the final SSE event on streams — carries a `usage` block the gateway *normalizes* to one schema:

```
{
  "input_tokens": 4120,
  "output_tokens": 356,
  "cached_input_tokens": 3800,
  "reasoning_tokens": 0,
  "modalities": { "images_in": 0, "images_out": 0 },
  "provider_model_id": "enterprise-gpt-4o-class-2026-01",
  "provider": "provider-c-enterprise",
  "latency_ms": { "tftb": 210, "total": 4820 }
}
```

The normalization layer is where multi-provider becomes real: each provider reports usage with different field names, some omit cached tokens, some bill reasoning tokens separately. The normalizer maps to the schema above with provider-specific adapters, and **the metering pipeline (2.6) consumes only the normalized object**. If a provider's response has no usage block, the gateway estimates from the serialized message (marked `estimated: true`) and the reconciliation job (Chapter 4) knows estimates exist.

- **Pass-through for the long tail:** `POST /ptp/{provider}/{any-path}` — authenticated, metered, audited, but body untouched. This is for batch APIs, fine-tuning, provider-specific features the canonical surface does not have. Pass-through still goes through auth, RPS, and token-budget checks; it simply does not enforce the alias rule. (Yes, that admits raw model IDs — scoped, flagged, and reserved for the long tail, because the alternative is a second uncontrolled path, which is worse.)
- **Idempotency:** `Idempotency-Key` header on non-streaming calls; the gateway stores request hash → response for 24h. Streaming calls are *not* idempotent (you cannot safely replay a half-stream); for those the contract is at-most-once with client-side retry policy, and the metering pipeline de-duplicates by provider request ID so a retried request that the provider charged twice is visible as two usage objects, not silently merged.

2.5 Routing / limits layer

The routing layer answers, in order: *who may do this? how much is left? who serves it? what if they cannot?*

1. **Alias resolution:** `model: standard` → binding → (provider, `model_id`). If the binding's provider is `degraded` per probes, fail over per policy: same-alias secondary binding first (e.g., `standard` also bound to a provider-B model at reduced rank), then deny with `503 model_unavailable` and a `Retry-After`. Fail-over is *visible*: the response header `X-LLM-Router: failed-over=1` and the metering event carry `routed_to != requested_binding`, so a cost report can say "X% of standard traffic ran on the secondary."
2. **Limits, in two layers:**
 - **RPS** (requests per second, token bucket per identity+role+alias): protects providers and your own queue.
 - **Token budget** (Chapter 4, summarized here): daily/weekly/monthly token and cost budgets per user, per team, per alias-class. The check runs *before* the provider call using the request's declared or estimated size; the *settlement* runs after, from the real usage object. This two-phase design matters: the pre-check admits on estimates, the settlement corrects on truth, and a team that repeatedly over-estimates (long-context abuse) shows up as a settlement variance metric before it shows up on the invoice.
3. **Priority and queueing:** at v1, no queue — over-limit is a fast `429` with `Retry-After` and a clear budget pointer ("you are at 91% of your monthly budget for alias `reasoning`"). Queueing is a v2 feature; the metering pipeline is already shaped to support it (the bus has per-tenant lanes), but a queue that hides pressure from the team is a cost problem with better UX.
4. **Circuit breaker per provider:** error-rate or latency-p95 breach on a provider for N seconds opens the breaker for that provider's *new* connections (in-flight streams finish), traffic shifts per fail-over policy,

and the breaker state is a first-class metric. This is the difference between "provider C is having a bad day" and "our services are melting down."

Go-style pseudocode for the hot path (deliberately flat; this is the whole loop):

```
func (g *Gateway) Handle(r *http.Request) {
    id, err := g.auth.Verify(r)          // JWT: user, team, role, scopes
    if err != nil { g.deny(r, err); return }

    req, err := g.decode(r)              // canonical or pass-through
    if err != nil { g.deny(r, err); return }

    if !g.auth.Allows(id, req.Alias, req.Mode) { g.deny(r, ErrRBAC); return }

    if err := g.limits.CheckRPS(id, req.Alias); err != nil { g.deny(r, ErrRPS); return }
    if err := g.limits.CheckBudget(id, req.Alias, req.Estimate()); err != nil {
        g.deny(r, err); return          // err carries budget % and owner contact
    }

    binding, over := g.registry.Resolve(req.Alias)
    resp, usage, err := g.provider.Call(binding, req, id) // streams back to client
    // metering: always, even on error paths (partial usage is still usage)
    g.meter.Emit(MeterEvent{ ID: id, Alias: req.Alias, RoutedTo: binding,
        Usage: usage, Est: req.Estimate(), Over: over, Err: err })
    g.audit.Emit(AuditEvent{ ID: id, Alias: req.Alias, Bytes: r.ContentLength,
        Outcome: outcome(err), TTFB: ... })
}
```

2.6 Metering / cost pipeline

The pipeline is: **capture** → **normalize** → **cost** → **roll up** → **publish**.

- **Capture**: every request produces exactly one `MeterEvent` (pseudocode above), including failures (a 500 from the provider that already consumed input tokens is an event with partial usage — providers bill it, so we must).
- **Normalize**: provider adapters → the canonical usage schema (2.4). Adapters are a per-provider file, unit-tested against recorded fixtures, and the test corpus *includes* a fixture for "usage block missing" per provider.
- **Cost**: $\text{cost} = \sum (\text{tokens} \times \text{price_per_mtok})$ over the line items (input, output, cached_input, reasoning, per-modality), using the price *booked* at request time (price is snapshotted into the event, never re-derived — this is what makes monthly reconciliation possible: you can replay any day's events against any day's prices).
- **Roll up**: stream consumers maintain:
 - **Hot counters** in Redis: per (user, team, alias) — last 5 min / 24 h / month — for the pre-check and the live dashboards;
 - **Facts** in SQL: one row per request (trimmed), retained hot for 90 days, then aged to cold (Chapter 7);

- **Rollup tables:** per day × (model, alias, team, user, feature) for dashboards and reconciliation.
- **Publish:** the rollups DB is the dashboard data source; no second "metrics lake" at v1.

The feature dimension: callers declare `X-LLM-Feature: code-review` (or the portal-registered feature name). Features are registered objects owned by a team lead; unregistered feature names are accepted but reported as `unregistered`, and the reconciliation report (4.7) lists spend by unregistered feature — the nudge that keeps the attribution axis honest.

2.7 Audit

Audit is a second, *separate* stream from metering, and the separation is the design: metering answers "what did it cost"; audit answers "what was done." Audit events:

```
{
  "ts": "2026-01-15T10:22:31.412Z",
  "actor": { "principal": "user:8842", "team": "payments", "role": "contributor",
    "on_behalf_of": null, "client": "service:code-review-bot" },
  "action": "llm.inference",
  "object": { "alias": "reasoning", "feature": "code-review" },
  "outcome": "ok",
  "bytes_in": 48211, "bytes_out": 2104,
  "policy": { "rps_bucket": "contributor:reasoning", "budget_check": "admit" },
  "correlation_id": "req_01J8ZK..."
}
```

Properties that matter: append-only, write-once to the audit stream; retention and access rules in Chapter 7; **payload-blind by default** — audit records shapes, not content (content logging is an explicit, separately-approved mode per endpoint class; the default posture is that request bodies never enter the audit stream). Auditor role access (3.5) reads audit, never metering's cost columns and never content.

2.8 Observability

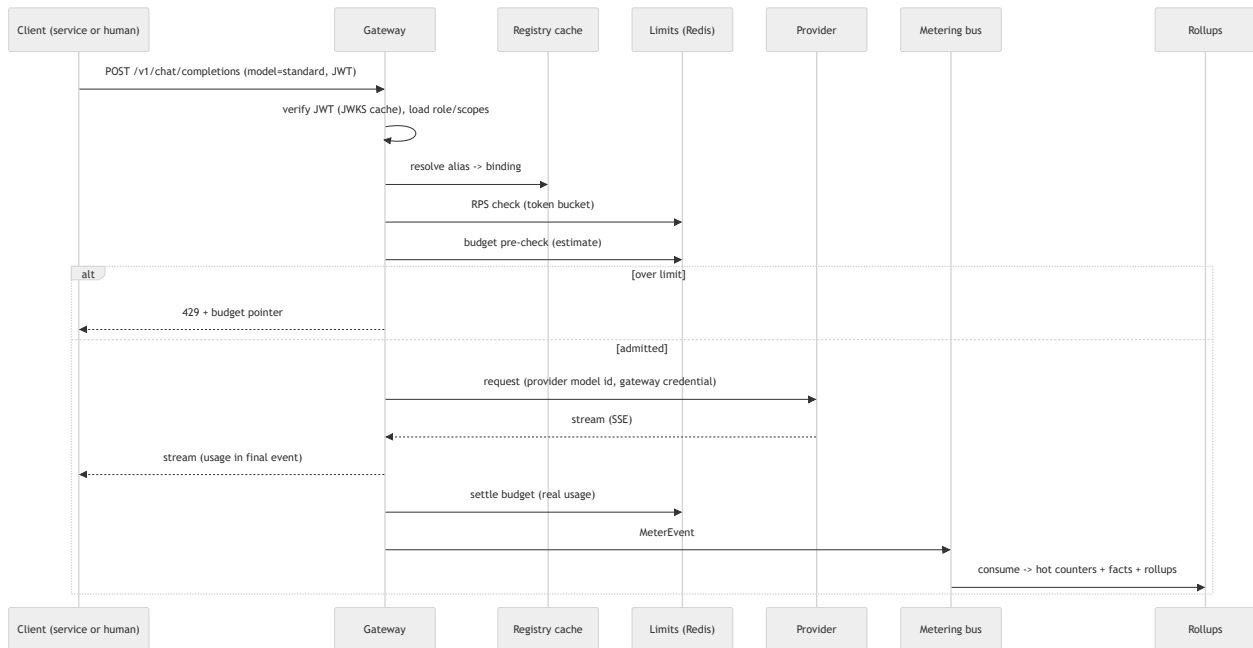
Three signals, deliberately minimal at v1:

- **Metrics** (from the gateway process): request rate / error rate / latency (p50/p95/p99, TTFB, total-stream) per (alias, provider, outcome); budget-admit/deny rates; breaker states; registry cache age; metering bus lag.
- **Traces:** one span per gateway hop + provider call, `correlation_id` propagated; the trace carries the *normalized usage* as a span event so a single trace answers "this request: N input tokens, M output, \$0.0214." Cost-per-trace is the single most useful debugging view this system has, and it is free once usage is on the span.
- **Logs:** structured JSON, one line per request (no bodies), fields mirroring the metering event so log queries and usage queries agree.

SLOs at v1 (tune, but start here): gateway availability 99.9% (monthly), TTFB p95 < 150ms *gateway-added* (provider latency excluded — measure both), metering completeness ≥ 99.9% of requests (a dashboard,

and an alert below 99.5%). The completeness SLO is what makes the cost numbers trustworthy to finance; it is the SLO the CFO's question depends on.

2.9 Request lifecycle (sequence)



Every arrow that carries data has a corresponding dashboard tile (Chapter 4) — if a step is not observable, it is not in the architecture.

2.10 Deployment shape

- **Proxy:** 3+ instances behind your LB (or the mesh's), horizontally scalable, no local state beyond caches; SSE-aware timeouts (per-provider, generous; your LB's idle timeout must exceed the longest expected stream — a classic silent failure, see pitfalls).
- **Redis:** the limits/counter tier; single managed instance with backup at v1 is fine (counters are recomputable; losing one causes over-admission for a TTL, not data loss).
- **SQL:** registry + facts + rollups; one DB at v1, schema-separated.
- **Bus:** your existing stream; two topics (`llm.metering.v1` , `llm.audit.v1`), audit topic with the retention policy from Chapter 7, not the default.
- **Identity:** the gateway's own workload identity (managed, keyless — Chapter 3) plus a JWKS endpoint for validating caller tokens.
- **laC:** the entire data plane is one module; environment parity is a config file. Chapter 7's deployment checklist is the gate between "works in my sandbox" and "deployed."

2.11 Failure modes, designed

Failure	Design response	Residual
Provider outage	Probes → breaker → fail-over binding or fast 503	All bindings down = your users see it; that is correct
Registry DB down	Cache serves reads for TTL (5 min); unknown alias fails closed	New bindings do not apply for $\leq$ TTL
Redis down	Fail open for RPS (alert), fail closed for budgets (deny with pointer) — budgets protect spend, RPS protects providers	Short over-admission window on RPS
Bus down	Bounded in-process queue (60s) + alert; on overflow, metering loss metric increments	Loss is measured, never silent
Gateway itself down	Degrade mode (7.2): scoped direct-provider keys continue at reduced control	Control gap during the incident; runbook closes it
Price data stale	Booked price snapshotted per event; drift alerts	Budgets run on stale price until refreshed — acceptable, flagged

The design goal visible in this table: **no failure mode is silent**. Every degradation is a metric, an alert, or both.

2.12 How to implement this in 90 days

Assumption: 2–4 engineers, existing cloud, existing identity platform, existing message bus. This plan has landed at the described scale; it is a shape, not a promise.

Days 1–15 — the pipe. Stand up the proxy skeleton: canonical `/v1/chat/completions` (streaming + non), one provider, one alias (`standard`), auth via your identity platform's workload tokens (no human keys yet, that is fine at this stage), metering event to the bus, usage normalization for that one provider, the one-line-per-request log. *Exit: one real service in your org calls `standard` through it, and its usage is in the rollups DB.*

Days 16–30 — the registry. Models/providers/aliases tables; the sync job for catalog + availability probes + price refresh on the first provider; `/v1/models` returns aliases; alias resolution + fail-over policy (even with one binding, the code path exists); the probe-tagged metering exclusion. *Exit: rename a model in the registry and watch bindings follow with zero caller changes.*

Days 31–45 — auth and RBAC (Chapter 3). Full keyless broker: no static provider key in any config; OBO flow for human callers; the six roles with per-role RPS and daily token budgets enforced; the two-phase budget check. *Exit: a contributor token can exceed no budget without a 429, and an auditor token can read nothing but audit.*

Days 46–60 — metering and dashboards (Chapter 4). Facts + rollups, the four dashboard views (live cost, budget burn, per-model, per-team), the alerting design (channel + webhook), the reconciliation job

v1 (usage vs. provider invoice, one provider). *Exit: finance can see this month's spend to the day, per team, and the reconciliation run for last month matches the provider invoice within the variance threshold.*

Days 61–75 — second provider + pass-through + audit. Second provider end-to-end (this is where the normalizer earns its keep); `/ptp` pass-through; audit stream separate from metering; retention policies; the circuit breaker and fail-over drill (induce a provider outage in staging; the fail-over is boring). *Exit: the fail-over drill is a screenshot in the ops wiki, not a memory.*

Days 76–90 — hardening, cutover, and the first migration. The deployment checklist (7.6) run for production; the migration checklist (7.7) executed for the first *real* team off direct-provider calls; SLOs live; the incident runbooks (7.4) printed and their first (planned) drill done. *Exit: one team migrated, its usage fully attributed, and the managed-gateway or direct-call path it left is in the process of being decommissioned.*

What is deliberately *not* in 90 days: MCP server (Chapter 5 — its own 90-day plan inside the chapter), the RAG assistant (Chapter 6), the web portal's self-service flows (a read-only portal ships in days 46–60; self-service lands after), queueing, and multi-region. Each of those is a chapter, not a stretch goal.

2.13 Pitfalls

- **Building a platform, not a gateway.** The reference is one deployable plus jobs. If your design doc has seven services in the hot path, you are not building this book's gateway; you are building a company. Split at the seams (2.2) only when a seam actually aches.
- **The LB idle timeout shorter than the stream.** The single most common silent failure: long streams die at the LB, the client sees a truncated response, the provider billed the full output, and your metering says "truncated" while the invoice says "complete." Set LB/ingress idle timeouts above your p99 stream duration *and* add a dashboard for mid-stream disconnects with and without a final usage event.
- **Trusting the client's declared model.** A caller that sends a raw provider model ID is bypassing the lockout firewall. Reject it, log it, and route the offender's team to the portal. The firewall is the firewall only if it is rejected in one place.
- **Metering only successes.** A provider 500 that consumed 40k input tokens is a 40k-token event. If your capture is `if err == nil`, your books are wrong by exactly your provider's error rate, and the reconciliation chapter will catch you.
- **Price re-derived at report time.** Cost must be snapshotted per event at request time. Re-deriving against today's price makes history un-replayable and reconciliation impossible; this is a data-model decision, and the wrong one is unrecoverable.
- **One provider in the registry from day one is fine; one provider in the *architecture* is not.** The normalizer, the fail-over code, and the multi-provider rollups must exist before the second provider, not after it — "add the second provider when we need it" is how the second provider costs a quarter.
- **The probe's cost is not zero.** Probe requests are real tokens. Tag them, exclude them from business cost, and budget them (they are trivial, but an untagged probe is a reconciliation variance with no explanation).

- **Registry as source of truth for *policy*, sync for *facts* — never the reverse.** If someone hand-edits a synced availability field and the next sync overwrites it, the field was not a policy field. When that fight happens, the fix is to add a policy column, not to stop the sync.