Universal CSP Model-as-a-Service Playground: architecture blueprint

A multi-provider AI gateway with a playground UI can be built on a proven pattern: a FastAPI proxy normalizing all providers to the OpenAI API format, envelope-encrypted credential storage, and a React-based streaming frontend. This architecture draws directly from production-validated projects—LiteLLM (100+ providers, Medium GitHub 8ms P95 latency), Portkey (10B+ requests/month), Portkey and Helicone (2B+ logged interactions)— Helicone while adding a purpose-built playground experience for developers and enterprise teams. The design prioritizes security isolation (per-org envelope encryption with KMS), horizontal scalability (stateless proxy pods behind K8s HPA), and real-time streaming (SSE normalization across all providers). Below is the complete architectural proposal with component design, technology choices, and phased implementation roadmap.

System architecture and component overview

The system follows a **three-tier architecture**: a React/Next.js frontend, a FastAPI gateway/API layer, and a data tier (PostgreSQL + Redis + object storage). The gateway acts as an OpenAI-compatible reverse proxy that intercepts requests, resolves user credentials, transforms payloads to provider-native formats, streams responses back, and logs everything asynchronously. Medium

```
┌──────────────────────────────────────────────────────────────────────────────
│                    FRONTEND (Next.js)                    │
│  ┌──────────────────┐  ┌──────────────────────┐  ┌──────────────────┐
┌──────────────────────────┐   │
│  │ Chat       │  │ Side-by-Side │  │ Prompt      │  │ Usage/Cost     │   │
│  │ Playground │  │ Comparison    │  │ Templates │  │ Dashboard      │   │
│  └───────────┬──────┘  └──────────┬───────────┘  └──────────┬───────┘
└───────────┬──────────────┘   │
│
└────────────────────────┬────────────────────────┬──────────────────────────┘
│
│                   SSE + REST API                    │
└──────────────────────────────────────────────┬───────────────────────────────
                      │ HTTPS
┌──────────────────────────────────────────────▼───────────────────────────────
│                API GATEWAY (FastAPI + Uvicorn)                │
│                                                │
│  ┌────────────────┐  ┌──────────────────────┐  ┌───────────────────┐
┌──────────────────────┐   │
│  │ Auth      │ → │ Rate Limiter │ → │ Router /   │ → │ Provider      │  │
│  │ Middleware │  │ (Redis)      │  │ Load       │  │ Abstraction  │  │
│  │ (JWT)     │  │            │  │ Balancer  │  │ Layer       │  │
│  └────────────────┘  └──────────────────────┘  └───────────────────┘
└──────────┬───────────┘   │
│                                │      │
│
┌──────────────────────────────────────────────────────────────────────────────
│
│  │        Provider Adapters (Plugin System)       │ │       │
│  │  ┌─────────────┐  ┌──────────────┐  ┌─────────────┐
┌─────────────────────┐  │ │      │
│  │  │ OpenAI   │  │ Anthropic │  │ Gemini  │  │ AWS Bedrock  │  │ │       │
│  │  │ Adapter  │  │ Adapter  │  │ Adapter │  │ Adapter      │ │ │       │
│  │  └─────────────┘  └──────────────┘  └─────────────┘
└─────────────────────┘  │ │      │
│  │  ┌─────────────┐  ┌──────────────┐  ┌─────────────┐
┌─────────────────────┐  │ │      │
│  │  │ Azure    │  │ Vertex AI │  │ Cohere  │  │ Mistral      │  │ │       │
│  │  │ Adapter  │  │ Adapter  │  │ Adapter │  │ Adapter      │ │ │       │
│  │  └─────────────┘  └──────────────┘  └─────────────┘
└─────────────────────┘  │ │      │
│  │  ┌─────────────┐  ┌──────────────┐  ┌─────────────┐           │ │      │
│  │  │ vLLM     │  │ Ollama    │  │ TGI     │          │ │      │
│   │   │ Adapter │   │ Adapter  │   │ Adapter │             │ │        │
│   │   └─────────────┘   └───────────────┘   └─────────────┘            │ │        │
│
└────────────────────────────────────────────────────────────────────────────
│
│                                  │       │
│  ┌──────────────────────────┐  ┌──────────────────────────┐
┌────────────────────────┐  │       │
│  │ Credential    │   │ Usage/Cost   │   │ Async Log    │ │        │
│  │ Decryptor     │   │ Calculator   │  │ Writer      │ │        │
│  │ (Vault/KMS)   │   │ (tiktoken)   │  │ (Redis Stream │ │        │
│  └──────────────────────────┘  └──────────────────────────┘
└────────────────────────┘  │       │
└────────────────────────────────────────────────────────────────────────────
              │
┌─────────────────────────────────────────────────────▼──────────────────────
│              DATA TIER                       │
│  ┌────────────────────────┐  ┌────────────────────────┐
┌──────────────────────────────────────────────┐ │
│  │ PostgreSQL    │   │ Redis 7+     │   │ HashiCorp Vault /        │ │
│  │ (Users, Orgs, │   │ (Cache, Rate │   │ AWS KMS                 │ │
│  │  Keys, Logs,  │   │  Limits,     │  │ (KEK management,        │ │
│  │  Usage)       │   │ Sessions,    │  │  envelope encryption)   │ │
│  │         │   │  Pub/Sub)    │  │                   │ │
│  └────────────────────────┘  └────────────────────────┘
└──────────────────────────────────────────────┘ │
│  ┌────────────────────────┐  ┌────────────────────────┐                  │
│  │ S3/Blob Store │   │ ClickHouse   │ (Phase 2: analytics at scale) │
│  │ (Archived     │   │ (OLAP for    │                   │
│  │  logs, audits │   │  usage data) │                   │
│  └────────────────────────┘  └────────────────────────┘                  │
└────────────────────────────────────────────────────────────────────────────
```

The gateway is **stateless by design**—all shared state lives in PostgreSQL and Redis, enabling horizontal pod autoscaling. A single request flows through: **authentication** → **rate limiting** → **credential decryption** → **request transformation** → **provider call** → **response transformation** → **streaming to client** → **async logging**.

Provider abstraction layer: the adapter pattern that works at scale

The provider abstraction layer is the architectural heart of the system. Every existing production gateway (LiteLLM, Portkey, OneAPI) converges on the same proven approach: an **adapter pattern with a unified base contract**, where each provider implements transformation methods to convert between the OpenAI-compatible format and its native API.

The base contract defines five responsibilities every adapter must fulfill:

```python
class BaseProviderAdapter(ABC):
    """Contract all provider adapters implement."""

    @abstractmethod
    def validate_environment(self, credentials: ProviderCredential) -> dict:
        """Build auth headers from decrypted credentials."""

    @abstractmethod
    def transform_request(self, model: str, messages: list, params: dict) -> dict:
        """OpenAI-format request → provider-native payload."""

    @abstractmethod
    def transform_response(self, raw_response: Any) -> ChatCompletionResponse:
        """Provider-native response → OpenAI-format response."""

    @abstractmethod
    def transform_streaming_chunk(self, chunk: str) -> dict:
        """Provider SSE chunk → OpenAI-format streaming chunk."""

    @abstractmethod
    def get_supported_params(self, model: str) -> list[str]:
        """Declare which OpenAI params this provider supports."""
```

Provider-specific features that don't exist in other providers are handled through three mechanisms: (1) unsupported parameters are silently dropped (Zread) when `drop_params=true` is configured, using the `get_supported_params()` declaration; (2) provider-unique features are passed through an `extra_body` extension field (e.g., Anthropic's `cache_control`, Gemini's `thinking_config`); (3) semantically equivalent features across providers are automatically translated (e.g., OpenAI's `reasoning_effort` maps to Gemini's `thinking_level`). (LiteLLM)

Mapping complexity varies significantly by provider. OpenAI is pass-through (reference format). Azure OpenAI requires only a different base URL and `api_version` parameter. Mistral is nearly OpenAI-compatible. Self-hosted models (vLLM, Ollama, TGI) expose OpenAI-compatible endpoints natively— (Medium) minimal transformation needed. **Anthropic requires medium effort**: different role handling (system messages become a top-level parameter), content structured as arrays of blocks. **AWS Bedrock is the most complex**: requires SigV4 request signing, uses its own Converse API format, and has distinct tool/message schemas. Google Vertex AI similarly requires OAuth2 authentication and its own endpoint structure, though Gemini now offers an OpenAI-compatible endpoint at `generativelanguage.googleapis.com/v1beta/openai/`. Google AI

The gateway exposes these unified endpoints, all following OpenAI's schema:

- `POST /v1/chat/completions` — Chat completions (primary)
- `POST /v1/embeddings` — Text embeddings
- `POST /v1/images/generations` — Image generation
- `POST /v1/audio/speech` — Text-to-speech
- `POST /v1/audio/transcriptions` — Speech-to-text
- `GET /v1/models` — List available models across all configured providers

A model is addressed by its **prefixed identifier**: `openai/gpt-4o`, `anthropic/claude-sonnet-4`, `bedrock/anthropic.claude-3-sonnet`, `ollama/llama3.1`. The prefix routes through a factory that instantiates the correct adapter.

Streaming architecture: SSE normalization across every provider

Streaming is critical for user experience—**time-to-first-token (TTFT) matters more than total latency** for perceived responsiveness. Pockit The gateway normalizes all provider streaming formats to OpenAI's SSE chunk format:

```
data: {"id":"chatcmpl-xxx","object":"chat.completion.chunk",
       "choices":[{"delta":{"content":"Hello"},"index":0}]}

data: {"id":"chatcmpl-xxx","object":"chat.completion.chunk",
       "choices":[{"delta":{"content":" world"},"index":0}]}

data: [DONE]
```

The proxy streaming pipeline works as follows: the FastAPI endpoint receives a `stream: true` request, opens an `httpx.AsyncClient` streaming connection to the provider, and returns a `StreamingResponse` that yields transformed chunks as an async generator. JunKangWorld **Critical configuration**: Nginx must have `proxy_buffering off` and `X-Accel-Buffering: no`, Pockit Proagenticworkflows and the outbound HTTP client needs a **120–300 second read timeout** (AI inference can be slow). For non-streaming long requests, SSE heartbeat comments (`:heartbeat\n\n`) every 15 seconds prevent intermediate proxies from timing out. Pockit

Token counting during streaming uses a dual-precision approach. During the stream, output tokens are estimated incrementally (roughly 1 token per 4 characters) for real-time UI cost display. After the stream completes, the provider's reported usage (from the final chunk or response headers) provides exact counts that reconcile the estimate. Input tokens are counted pre-stream using **tiktoken** Towards Data Science (for OpenAI-family models) or provider-specific tokenizers (Anthropic's API, Gemini's `count_tokens()`). LiteLLM's `token_counter()` auto-selects the correct tokenizer by model name. LiteLLM

Error handling during streaming requires careful design. If an error occurs before the first chunk, a standard HTTP error response is returned. If an error occurs mid-stream, an SSE error event is emitted followed by `[DONE]`, since HTTP status codes cannot be changed after headers are sent. DeepWiki The client must handle partial responses gracefully.

API key management: envelope encryption with per-org isolation

Secure credential storage is the highest-stakes security concern. Users entrust the platform with API keys that may have significant spending authority. The architecture uses **envelope encryption with a managed KMS as root of trust**—the same pattern AWS Secrets Manager uses internally. AWS

The encryption flow for storing a user's API key:

1. Generate a unique **Data Encryption Key (DEK)** per credential using a CSPRNG (256-bit AES key)
2. Encrypt the provider API key with the DEK using **AES-256-GCM** (provides confidentiality + integrity via authentication tag)
3. Wrap the DEK using a **Key Encryption Key (KEK)** managed by HashiCorp Vault Transit or AWS KMS
4. Store both the encrypted credential and the encrypted (wrapped) DEK in PostgreSQL; discard the plaintext DEK from memory immediately
5. To decrypt at request time: send wrapped DEK to KMS → receive plaintext DEK → decrypt credential locally → use → discard

Per-organization KEK isolation ensures that a compromise of one org's data cannot affect another. The hierarchy runs: Root KMS Key → per-organization KEK → per-credential DEK. Key rotation is handled by Vault's `rewrap` command—re-encrypts DEKs under the latest key version without touching underlying credential data. HashiCorp Developer KodeKloud Notes

Different credential types demand different handling:

Provider	Credential Type	Storage Approach
OpenAI, Anthropic, Cohere, Mistral	Bearer API key	Encrypt single string
AWS Bedrock	IAM Access Key + Secret Key	Encrypt both; prefer cross-account IAM role assumption (store only Role ARN)
GCP Vertex AI	Service Account JSON key	Encrypt full JSON; prefer Workload Identity Federation
Azure OpenAI	API key or Service Principal	Encrypt all components as a bundle

For security-conscious users, the platform offers a **pass-through mode** where the provider key is sent in each request header and never persisted server-side. This sacrifices server-side retry and caching capabilities but provides zero-knowledge guarantees.

Backend technology stack: why FastAPI wins for AI gateways

Python 3.11+ with FastAPI is the clear technology choice, validated by every major open-source AI gateway. LiteLLM, the dominant gateway with 100+ providers, is built on FastAPI. Medium AWS's official Multi-Provider GenAI Gateway reference architecture uses LiteLLM GitHub on FastAPI. Aws-solutions-library-samp··· The reasoning is decisive:

- **AI ecosystem dominance**: Every provider SDK is Python-first. Anthropic, OpenAI, Google, Cohere—all publish Python as the primary SDK. Building in Node.js means fighting the ecosystem.
- **Async-native**: FastAPI on Uvicorn provides native `async/await` for the I/O-bound workload of proxying API calls. A single Uvicorn worker can handle hundreds of concurrent provider connections.
- **SSE streaming**: `StreamingResponse` with async generators provides first-class SSE support. JunKangWorld Medium
- **Type safety**: Pydantic models provide request/response validation with automatic OpenAPI documentation.
- **Performance**: **8ms P95 latency at 1,000 RPS** demonstrated by LiteLLM in production. GitHub The bottleneck is provider latency (seconds), not gateway overhead (milliseconds).

The production deployment stack:

```
Client → Nginx/ALB (SSL termination, proxy_buffering off)
    → Gunicorn (process manager)
    → Uvicorn workers (1 per CPU core, ASGI)
    → FastAPI application
    → httpx.AsyncClient (outbound, connection-pooled per provider)
```

A critical implementation detail: the outbound `httpx.AsyncClient` must be a singleton per provider (not instantiated per request) to leverage TCP connection pooling and HTTP/2 multiplexing. Default limits of 100 max connections and 20 keepalive connections (HTTPX) per provider handle most workloads.

Component	Technology	Justification
Language	Python 3.11+	AI ecosystem, all SDKs Python-first
Framework	FastAPI + Uvicorn	Async-native, SSE, Pydantic, proven by LiteLLM
HTTP Client	httpx.AsyncClient	Connection pooling, HTTP/2, streaming
Database	PostgreSQL 16+	ACID, JSONB, Row-Level Security for multi-tenancy
Cache/Rate Limiter	Redis 7+ Cluster	Sliding window rate limits, session cache, response cache
Message Queue	Redis Streams (→ Kafka at scale)	Async log writing, usage aggregation
Secret Management	HashiCorp Vault Transit	Cloud-agnostic KEK management, key rotation
Frontend	Next.js 15 + shadcn/ui + assistant-ui	SSR, streaming API routes, best AI chat component ecosystem

---

Web UI playground: chat, compare, and tune

The playground frontend serves two core workflows: **single-model conversation** (ChatGPT-style) and **multi-model comparison** (side-by-side evaluation). The interface follows a three-panel layout validated by OpenAI Playground, Vercel AI Playground, and Google AI Studio.

```
┌──────────────┬──────────────────────────────────┬──────────────────
│  Sidebar     │  Chat / Comparison Area          │  Settings       │
│              │                                  │  Panel          │
│ • Chat       │  ┌───────────┬───────────┐       │                 │
│   History    │  │ Model A   │ Model B   │       │  Model: [▼]     │
│ • Folders    │  │ GPT-4o    │ Claude    │       │  Temp: ───●     │
│ • Prompt     │  │           │           │       │  Top P: ──●     │
│   Templates  │  │ Hello,    │ Hello!    │       │  Max: [4096]    │
│ • Search     │  │ world!    │ I'm       │       │                 │
│              │  │           │ Claude.   │       │  System Prompt  │
│ + New Chat   │  └───────────┴───────────┘       │  ┌───────────┐  │
│              │                                  │  │ You are...│  │
│              │  ┌──────────────────────────┐    │  └───────────┘  │
│              │  │ [📎] Type message...  ▶  │    │                 │
│              │  └──────────────────────────┘    │  Cost: $0.003   │
│              │  Tokens: 847 | Cost: $0.01       │  TTFT: 0.4s     │
└──────────────┴──────────────────────────────────┴──────────────────
```

The **model comparison mode** sends the same prompt to 2–3 models in parallel, streaming responses independently with per-panel latency and cost metrics. After comparison, users can "Continue chat with Model X" to select a winner and proceed in single-model mode. This is the killer feature for the target audience—developers evaluating which model fits their use case.

Key frontend technology choices: **Next.js 15** for SSR and streaming API routes. **assistant-ui** (50K+ monthly downloads, YC-backed) provides composable AI chat primitives built on Radix/shadcn patterns—auto-scroll, streaming display, accessibility. **shadcn/ui** with Tailwind CSS for the design system. **react-markdown** with remark-gfm, rehype-highlight, and KaTeX for rendering model responses with code highlighting, LaTeX, and Mermaid diagrams.

Prompt template management follows the pattern established by Langfuse and LangSmith: templates use `{{variable}}` placeholders, carry version history with git-like commit hashes, support labels ("production", "staging"), and can be pulled via API with version pinning. Langchain Templates store associated model configuration (model, temperature, max_tokens) alongside the prompt text. Langchain

---

Rate limiting, retry, and fallback: building resilience

The rate limiting architecture operates at three layers:

1. **Per-user/per-key limits** (gateway-enforced): Requests per minute (RPM) and tokens per minute (TPM) tracked in Redis using sliding window counters

2. **Per-provider limits** (respecting upstream quotas): The gateway reads provider rate limit headers (`x-ratelimit-remaining-requests`, `x-ratelimit-remaining-tokens`) to proactively slow down before hitting hard limits
3. **Per-deployment limits** (multiple API keys per provider): When a user registers multiple keys for the same provider, the router distributes load across them—effectively multiplying rate limits

Retry logic uses exponential backoff with full jitter: Openai `delay = base_delay × 2^attempt × random(1, 2)`, respecting the provider's `Retry-After` header when present. Maximum **3 retries** within the same model group before escalating to fallback.

Fallback chains support two modes: **intra-model fallback** (same model across different deployments—e.g., GPT-4o via OpenAI direct and Azure OpenAI) and **cross-model fallback** (e.g., GPT-4o → Claude Sonnet → Gemini Pro). Portkey A **circuit breaker** per deployment tracks failures: after **5 consecutive failures**, the deployment enters a 60-second cooldown where it receives no traffic. After cooldown, a single probe request tests recovery. Medium

Routing strategies (configurable per user or per route):

- **Simple shuffle**: Random distribution across healthy deployments (default) LiteLLM
- **Least-busy**: Route to deployment with fewest active requests
- **Usage-based**: Route based on TPM/RPM consumption tracking
- **Latency-based**: Route to lowest-latency deployment
- **Cost-based**: Route to cheapest provider first

Usage tracking, cost estimation, and dashboards

Cost visibility is a first-class feature. The system maintains a **pricing database** (JSON/YAML) covering 400+ models with input/output cost per million tokens, updated daily from LiteLLM's live pricing API and provider announcements. Each request generates a cost record: `input_tokens × input_price + output_tokens × output_price`.

The **usage dashboard** presents five core views:

- **Overview cards**: Total requests, tokens, cost, and active users for the selected period
- **Cost trend chart**: Time-series stacked area showing cost by provider (hourly/daily/weekly)
- **Model breakdown**: Donut chart of cost distribution across models, revealing which models consume budget

- **Cost comparison table**: After running a prompt through multiple models, a side-by-side table shows input cost, output cost, total cost, and latency per model—the most actionable view for model selection decisions
- **Budget alerts**: Per-user and per-team daily/monthly thresholds with soft warnings (in-app + email) and hard limits (requests blocked)

Pre-aggregated `usage_daily` tables (org_id, date, provider, model, request_count, tokens, cost) power the dashboards without scanning the full request log table. For deployments exceeding **10M requests/month**, ClickHouse provides columnar analytics over historical data archived from PostgreSQL.

Monitoring, observability, and the LGTM stack

The observability architecture follows the **LGTM stack** (Loki, Grafana, Tempo, Mimir) unified under Grafana dashboards:

- **Metrics**: Prometheus scrapes the FastAPI `/metrics` endpoint exposing `llm_requests_total`, `llm_request_duration_seconds` (histogram for p50/p95/p99), `llm_time_to_first_token_seconds`, `llm_tokens_total`, `llm_cost_usd_total`, and `llm_errors_total` —all labeled by model, provider, user, and team (LiteLLM)
- **Traces**: OpenTelemetry SDK with OpenLLMetry auto-instrumentation captures per-request spans including prompt tokens, completion tokens, latency, cost, and model parameters. (Agenta) (Grafana) Exported to Grafana Tempo via OTLP
- **Logs**: Structured JSON request logs (request_id, provider, model, tokens, latency, status, cost) shipped to Grafana Loki—cost-efficient label-based indexing (Medium) maps perfectly to LLM metadata dimensions
- **LLM-specific observability**: Langfuse integration (open-source, OTel-compatible) provides prompt management, evaluation workflows, and session-level cost tracking (Langfuse) (Mirascope)

Critical alerts trigger on: provider error rate >5% for 5 minutes, p99 latency >30s for 10 minutes, rate limit remaining <10%, budget utilization >80%, or complete provider outage (zero successful requests for 2 minutes).

The **provider health dashboard** shows real-time status indicators (healthy/degraded/down) per provider, computed from a sliding 5-minute window of error rates and latency percentiles. A background health checker pings each provider every 30 seconds with a lightweight request.

Security, multi-tenancy, and compliance

Authentication uses OAuth 2.0/OIDC with short-lived JWT access tokens (15–30 minute expiry, RS256 asymmetric signing). The web UI authenticates via secure HttpOnly cookies; the API uses Bearer tokens. Curity Enterprise SSO supports SAML 2.0 and OIDC per-organization configuration via WorkOS or Keycloak. Requesty Platform API keys follow the format `gw_live_` + 32 bytes cryptographic random (base62-encoded); only the SHA-256 hash is stored in the database.

RBAC is scoped per-organization with four roles: Owner (full control, billing, SSO), Admin (members, keys, credentials, all logs), Member (use gateway, own logs, own credentials), and Viewer (read-only). Granular permissions (`credentials:write`, `logs:read`, `team:manage`) are grouped into roles and enforced in middleware. Mulesoft

Multi-tenant isolation uses PostgreSQL Row-Level Security (RLS). Every tenant-scoped table has an RLS policy enforcing `org_id = current_org_id()`, set at connection start via `SET app.current_org_id`. QuantumByte Nile This provides defense-in-depth: even if application code misses a WHERE clause, the database prevents cross-tenant data access. QuantumByte Redis keys are prefixed with `org:{org_id}:` to prevent cache pollution.

Audit logging captures authentication events, credential management actions, API requests (metadata only—not prompt content by default), admin actions, and system events. Prompt/completion content logging is **opt-in per organization** with explicit consent, and PII detection runs before storage when enabled. Logs are written to append-only storage (S3 with Object Lock) with configurable retention (minimum 90 days, 1+ year for enterprise). DigitalAPI

GDPR compliance requires regional deployments (EU, US gateway instances), geo-fenced provider routing (EU tenants' requests route only to EU provider endpoints), KEKs for EU tenants stored in EU KMS regions, Portkey Lasso and data export/deletion workflows supporting the right to erasure.

Database schema: core entities and scaling strategy

The schema centers on six core entities with **time-partitioned log tables** for performance at scale:

Organizations store tenant identity, plan tier, settings (JSONB), and data region. **Users** link to orgs through a many-to-many **OrgMemberships** table carrying role and permissions. **PlatformAPIKeys** store the SHA-256 hash of issued keys with scopes, rate limits, IP allowlists, and expiration. **ProviderCredentials** store the encrypted credential blob, encrypted DEK, KMS key reference, and a credential hint (last 4 characters for UI display).

RequestLogs are partitioned monthly by `created_at` using PostgreSQL native partitioning. Each record captures: org_id, user_id, api_key_id, credential_id, provider, model, prompt/completion tokens, latency_ms, status_code, estimated_cost, and metadata (JSONB). **UsageDaily** provides pre-aggregated summaries (org_id, date, provider, model, request_count, tokens, cost) with a unique constraint enabling `INSERT ... ON CONFLICT DO UPDATE` upserts.

Critical indexes for hot-path performance:

- `platform_api_keys(key_hash) WHERE is_active = TRUE` — every API request hits this
- `provider_credentials(org_id, provider) WHERE is_valid = TRUE` — credential lookup
- `request_logs(org_id, created_at DESC)` — primary log query with partition pruning
- `usage_daily(org_id, date DESC)` — dashboard queries

Logs older than 30–90 days archive to S3 in Parquet format, queryable via ClickHouse or Athena for historical analytics.

Deployment architecture: containers, scaling, and infrastructure

The deployment runs on **Kubernetes** with stateless application pods and managed data services:

```
┌── Ingress (Nginx/ALB, SSL, proxy_buffering=off) ──────────────────┐
│                                            │
│   ┌── Gateway Pods (HPA: 2-20 replicas) ────────────────────────┐    │
│   │  FastAPI + Uvicorn (1 worker/pod, async)          │    │
│   │  CPU target: 70% | Scale trigger: p95 latency > 100ms │    │
│
└──────────────────────────────────────────────────────────────────────
│
│                                            │
│   ┌── Worker Pods (HPA: 1-5 replicas) ──────────────────────────────┐    │
│   │  Async log writer, usage aggregation, webhooks      │    │
│   │  Consumes from Redis Streams                        │    │
│
└──────────────────────────────────────────────────────────────────────
│
│                                            │
│   ┌── Frontend (CDN-served static SPA or Next.js pods) ──────┐    │
│   │  CloudFront/Vercel/Cloudflare Pages                 │    │
│
└──────────────────────────────────────────────────────────────────────
│
│                                            │
│   ┌── Data Services
──────────────────────────────────────────────────────────────────┐    │
│   │  PostgreSQL (RDS/CloudSQL) + PgBouncer              │    │
│   │  Redis 7+ Cluster (ElastiCache/Memorystore)         │    │
│   │  Vault (self-hosted or HCP Vault)                   │    │
│   │  S3 (log archival, blob storage)                    │    │
│
└──────────────────────────────────────────────────────────────────────
│
└──────────────────────────────────────────────────────────────────────
```

Gateway pods are **stateless**—all shared state lives in PostgreSQL and Redis. HPA scales based on CPU utilization (target 70%) or p95 latency threshold. PodDisruptionBudgets ensure at minimum 2 pods during rolling updates. NetworkPolicies restrict pod-to-pod communication. The External Secrets Operator syncs KMS/Vault secrets into Kubernetes Secrets.

Infrastructure-as-code uses **Terraform/OpenTofu** with cloud-specific modules for managed services (RDS, ElastiCache, KMS on AWS; Cloud SQL, Memorystore, Cloud KMS on GCP). The core application is cloud-agnostic—only the data service adapters change per cloud.

Implementation roadmap: MVP in 6 weeks, full platform in 16

Phase 1 — MVP (weeks 1–6)

The MVP delivers a working playground with 5 providers and core functionality:

- **Backend**: FastAPI gateway with adapters for OpenAI, Anthropic, Google Gemini, Mistral, and Ollama. OpenAI-compatible `/v1/chat/completions` endpoint with streaming. Basic JWT auth with email/password registration.
- **Credential management**: AES-256-GCM encryption with a single application-level KEK (upgrade to Vault in Phase 2). CRUD UI for managing provider API keys.
- **Frontend**: Single-model chat playground with streaming, parameter tuning sidebar (temperature, top_p, max_tokens), model selector, conversation history (stored in PostgreSQL).
- **Cost tracking**: Token counting with tiktoken, basic per-request cost calculation, usage summary page.
- **Rate limiting**: Per-user RPM limits in Redis.
- **Deployment**: Docker Compose for local dev; single-node deployment with PostgreSQL + Redis.

Phase 2 — Core platform (weeks 7–12)

- **Providers**: Add AWS Bedrock, Azure OpenAI, GCP Vertex AI, Cohere, vLLM, TGI adapters.
- **Comparison mode**: Side-by-side 2–3 model comparison with parallel streaming.
- **Organizations**: Multi-tenancy with org/team hierarchy, RBAC (Owner/Admin/Member/Viewer), PostgreSQL RLS.
- **Security upgrade**: HashiCorp Vault Transit for KEK management, per-org key isolation.
- **Prompt templates**: Template editor with variables, versioning, team sharing.
- **Retry/fallback**: Configurable fallback chains, exponential backoff, circuit breakers.
- **Usage dashboards**: Cost trend charts, model breakdown, budget alerts.
- **Deployment**: Kubernetes with HPA, managed PostgreSQL and Redis.

Phase 3 — Enterprise features (weeks 13–16)

- **SSO**: SAML 2.0 and OIDC integration via WorkOS/Keycloak.
- **Observability**: Full LGTM stack (Prometheus, Grafana, Loki, Tempo), OpenTelemetry integration, provider health dashboard.

- **Advanced routing**: Latency-based, cost-based, and usage-based routing strategies. Load balancing across multiple keys per provider.
- **Audit logging**: Immutable audit trail with S3 Object Lock, configurable retention, SIEM export.
- **Data residency**: Regional deployments (EU/US), geo-fenced provider routing.
- **Response caching**: Exact-match and semantic caching for repeated prompts.
- **Embeddings and multi-modal**: `/v1/embeddings`, `/v1/images/generations`, `/v1/audio` endpoints.
- **API access**: Platform API keys for programmatic gateway access (not just playground UI).

Conclusion

The strongest architectural decision in this design is **leveraging LiteLLM's adapter pattern and provider abstraction as a dependency or reference implementation** rather than building provider integrations from scratch. LiteLLM's 100+ provider adapters represent thousands of hours of edge-case handling that would be impractical to replicate. The custom value-add sits above this layer: the playground UI, multi-tenant credential management with envelope encryption, cost comparison workflows, and enterprise security controls.

Three non-obvious insights emerged from the research. First, **the OpenAI Chat Completions format has become the universal LLM API standard**—even Google Gemini and Mistral now offer OpenAI-compatible endpoints natively, which means the abstraction layer's complexity will decrease over time, not increase. Second, **the data plane / control plane split** (pioneered by Portkey) is the optimal enterprise architecture: the gateway proxy runs in the customer's VPC (data never leaves their infrastructure) while management UI and configuration run as SaaS. Third, **cost observability is the most underestimated feature**—every production gateway project (Helicone, Portkey, LiteLLM) reports that cost tracking drives adoption more than routing or reliability features, because organizations cannot manage what they cannot measure.

The MVP can ship in 6 weeks with a small team (2 backend, 1 frontend), covering the primary use case: a developer registers their API keys, tests prompts across OpenAI/Anthropic/Gemini in a streaming playground, and sees what each request costs. Everything else—enterprise multi-tenancy, advanced routing, full observability—builds incrementally on this foundation without architectural rewrites.