GPU Cloud Platform

Architecture Documentation

1. System Overview

The platform is a mini AI PaaS that exposes an **OpenAI-compatible** chat completions API behind API key auth, rate limiting, and usage tracking. It uses a **distributed task queue** (Bull/BullMQ + Redis) to distribute inference work across multiple worker processes and stores API keys, deployments, and usage in **PostgreSQL**.

Component Diagram

```
                    ■■■■■■■■■■■■■■■■■■■■■■■■
                    ■   Frontend     ■
                    ■   (Next.js)    ■
                    ■   :3000        ■
                    ■■■■■■■■■■■■■■■■■■■■■■■■
                              ■ HTTP
                              ▼
■■■■■■■■■■■■■■■■■■■■     ■■■■■■■■■■■■■■■■■■■■■■■■     ■■■■■■■■■■■■■■■■■■■■
■  PostgreSQL  ■■■■■■■■■   API Server     ■■■■■■■■■    Redis      ■
■   :5432      ■      ■   (Express)     ■     ■   :6379      ■
■■■■■■■■■■■■■■■■■■■■    ■   :4000         ■     ■■■■■■■■■■■■■■■■■■■■
                     ■■■■■■■■■■■■■■■■■■■■■■■■■              ■
                               ■                       ■ Queue
                               ■                       ▼
                               ■              ■■■■■■■■■■■■■■■■■■■■■
                               ■              ■   Workers     ■
                               ■              ■   (BullMQ)    ■
                               ■              ■   N processes ■
                               ■              ■■■■■■■■■■■■■■■■■■■■■
                               ■                      ■
                               ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
                                     Usage metrics
```

Data Flow

Request → Client sends POST /v1/chat/completions with Authorization: Bearer <api_key>.

API Gateway → Express: CORS, JSON body, then middleware chain.

Auth → authenticateApiKey: load key by prefix, bcrypt compare, attach req.apiKey.

Rate limit → Redis sliding window: rate:{apiKeyId}:{window} INCR + EXPIRE; reject if over limit.

Validation → validateChatCompletionRequest: model, messages array, roles.

Inference → Controller either:

- **Non-streaming**: Call mockInference() in-process, record usage, return JSON.
- **Streaming**: Call mockInference(), then write SSE chunks (data: {...}, data: [DONE]).

Usage → recordUsage() inserts into usage_metrics; recordApiKeyUsage() bumps last_used_at and usage_count on api_keys.

The **queue** is used when you want to offload work to separate worker processes: jobs are added with addInferenceJob() and workers run Worker('inference', handler). The current HTTP path does in-process mock inference for low latency; the worker is available for scaling out.

2. Distributed Design

Queue pattern

BullMQ with Redis: one queue 'inference', job data includes requestId, model, messages, apiKeyId, stream.

Workers: 2–3 (or more) Node processes run the same worker script; each pulls jobs from the queue. Concurrency per worker is configurable (e.g. 4).

Retries: 3 attempts with exponential backoff (1s, 2s, 4s). Failed jobs can be moved to a dead-letter flow by listening to 'failed' and re-adding elsewhere.

Idempotency: Job ID = requestId so duplicate submissions don't create duplicate jobs.

Worker coordination

No leader election: all workers are equal consumers.

Redis handles atomic pop and locking; BullMQ manages retries and delays.

For 'wait for result' in HTTP, you'd typically use job.getState() / job.waitUntilFinished() or a separate result store (e.g. Redis by requestId) that the worker writes to and the API polls or subscribes to.

Failure handling

DB down: Health check fails; API returns 503 on /health; requests that need DB (auth, usage) fail with 500.

Redis down: Rate limiter fails open (next()); queue health fails; job adds would throw.

Worker crash: BullMQ marks job failed and retries per policy; after max attempts, job stays in failed set for inspection or DLQ.

3. Scalability

Current limits (single box)

~100 concurrent requests if each holds a DB connection and does in-process inference.

API key validation <100 ms with indexed key_prefix and bcrypt compare.

Queue latency <500 ms for Redis and a few workers on the same host.

Scaling to 1000 workers / high RPS

Redis: Move to **Redis Cluster** or a managed Redis with high connection and throughput limits; keep queue and rate-limit keys sharded or single-queue per region.

PostgreSQL: **Read replicas** for key validation and metrics reads; primary for writes (keys, usage). Connection pooling (e.g. PgBouncer) in front.

Workers: Run 1000 worker processes across many nodes; all consuming the same queue(s). Optionally **multiple queues** (e.g. by region or model) to isolate blast radius and prioritize.

API: **Horizontal scaling** of Express behind a **load balancer**; stateless, so no sticky sessions. Health checks for LB to remove bad nodes.

Frontend: **CDN** for static assets; optional serverless or cached dashboard that reads from the same API.

Geographic distribution

Deploy API + workers + Redis + DB per region. Use **global load balancing** (e.g. latency-based) to route to nearest region.

Cross-region: replicate DB (e.g. read replicas in other regions) or keep usage regional and aggregate in a central analytics DB.

4. Trade-offs

Decision	Choice	Why
Queue	Bull/BullMQ	Simple, Redis-based, good for 10k–100k jobs/day; no need for Kafka's throughput and op
DB	PostgreSQL	ACID, good for keys and time-series usage; simple to run and replicate. DynamoDB woul
Auth	API key only	Fits "server-to-server" and SDK usage; no OAuth/SSO in scope.
Inference	Mock in-process	Simulates GPU path; swap mockInference() for real model calls (local or remote).
Rate limit	Redis counter	Sliding window per key; sufficient for per-key limits. For global limits, add a separate globa

5. Future Work

Multi-tenancy: Namespace keys and usage by tenant_id; enforce quotas per tenant.

Cost tracking: Store cost per token or per model; aggregate in usage and expose in dashboard and billing.

Auto-scaling: Scale worker count (e.g. K8s HPA) based on queue depth or latency.

Real GPU: Replace mockInference() with calls to GPU inference services (e.g. Triton, vLLM, or external APIs) and optional streaming from worker to API via Redis or WebSocket.