CLOUD AI DEVELOPER GUIDE SERIES

Guide #5

# Production Architecture Patterns

Async Workflows · Multi-Cloud Failover · API Gateway Design
SLA Engineering · Cost Optimisation · Scalability Patterns


**Senior / Architect** ~35 min read Tier 3 of 3

cloudai.dev · 2025 Edition

## Table of Contents

# 01 Architecture Decision Framework

## Sync vs. Async — The Core Decision

Every AI integration starts with one question: does the user need to wait for the response, or can the work happen in the background? This single decision shapes your entire architecture. The 29-second limit of most API Gateways forces async for any non-trivial LLM workflow.

| Factor | Synchronous | Asynchronous |
|---|---|---|
| API Gateway limit | 29 sec hard limit | No limit — polling decouples |
| UX pattern | Inline response | Loading state + polling / webhook |
| Max LLM tokens | ~2K output (fast models) | Unlimited — Step Functions / queues |
| Error handling | Immediate retry | Dead-letter queue + replay |
| Cost profile | Predictable per-request | Cheaper at scale — batch + queue |
| Complexity | Low | Medium — state management needed |
| Best for | Chat, Q&A;, classification | Reports, pipelines, long generation |

## Token Rate Limits & Throttling

Every cloud AI platform enforces rate limits at two levels: **requests per minute (RPM)** and **tokens per minute (TPM)**. Hitting either limit returns HTTP 429. Your architecture must handle this gracefully.

| Platform / Model | Default RPM | Default TPM | Increase Path |
|---|---|---|---|
| Bedrock Claude 3 Haiku | 1,000 | 100,000 | Service quota increase via console |
| Bedrock Claude 3.5 Sonnet | 500 | 100,000 | Service quota increase via console |
| Azure GPT-4o (default) | 500 | 450,000 | PTU (Provisioned Throughput Units) |
| Azure GPT-4o mini | 1,000 | 1,000,000 | PTU or quota increase request |
| Vertex Gemini 1.5 Flash | 1,000 | 4,000,000 | Quota increase via Cloud Console |
| Vertex Gemini 1.5 Pro | 360 | 4,000,000 | Quota increase via Cloud Console |

**■■ Design for 429 from day one**

Do not treat rate limit errors as edge cases. At production scale they are routine. Implement exponential backoff with jitter on every LLM client. Use a token bucket / leaky bucket algorithm at the application layer to smooth traffic before it reaches the LLM API.

## Cold Start & Warm Pool Strategies

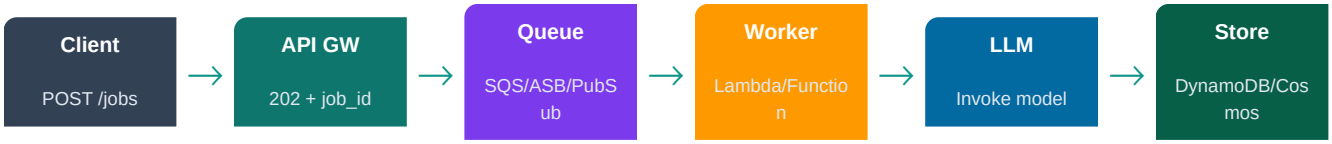
| | |
|---|---|
| **Lambda cold start** | First invocation of an idle Lambda container takes 200ms–2s. For LLM wrappers, use Provisioned Concurrency (AWS), Premium plan (Azure Functions), or min-instances (Cloud Run) to keep containers warm. |
| **Connection reuse** | Initialise the boto3/OpenAI/Vertex client outside the handler function so it is reused across warm invocations. A fresh HTTPS handshake adds 80-200ms per call. |
| **Model warm-up** | Some endpoints (especially self-hosted or fine-tuned models) have a model-load cold start. Use a scheduled keep-alive ping every 5 minutes if your SLA requires sub-second P99. |
| **Streaming responses** | Use streaming (SSE / WebSocket) for interactive chat to show the first token in <500ms even when total generation takes 10-30s. The user perceives much lower latency. |

## 02 Async Workflow Patterns

API Gateways enforce a hard 29-second (AWS) or 30-second (Azure/GCP) timeout. Any pipeline involving multiple LLM calls, agent loops, or RAG retrieval will routinely exceed this. The solution: decouple request acceptance from response generation.

### Pattern A — Async Submit + Poll

Best for: long-running pipelines, multi-agent workflows, batch generation




**Flow:** Client POSTs → API GW returns 202 with job_id instantly → Worker processes asynchronously → Client polls GET /jobs/{id} until status = COMPLETE

AWS Implementation (Lambda + SQS + DynamoDB + API Gateway):

```
import boto3, json, uuid, time

sqs = boto3.client("sqs")
dynamodb = boto3.resource("dynamodb")
table = dynamodb.Table("ai-jobs")
QUEUE_URL = "https://sqs.us-east-1.amazonaws.com/123456789/ai-jobs"

# --- SUBMIT HANDLER (API Gateway POST /jobs) ---
def submit_handler(event, context):
job_id = str(uuid.uuid4())
body = json.loads(event["body"])

# Write PENDING status immediately
table.put_item(Item={"job_id": job_id, "status": "PENDING",
"created_at": int(time.time())})

# Enqueue the actual work
sqs.send_message(
QueueUrl=QUEUE_URL,
MessageBody=json.dumps({"job_id": job_id, "prompt": body["prompt"]}),
)
# Return 202 Accepted instantly — well within 29s timeout
return {"statusCode": 202,
"body": json.dumps({"job_id": job_id, "status": "PENDING"})}

# --- WORKER HANDLER (triggered by SQS) ---
def worker_handler(event, context):
bedrock = boto3.client("bedrock-runtime", region_name="us-east-1")
for record in event["Records"]:
```

```
msg = json.loads(record["body"])
job_id = msg["job_id"]
table.update_item(
Key={"job_id": job_id},
UpdateExpression="SET #s = :s",
ExpressionAttributeNames={"#s": "status"},
ExpressionAttributeValues={":s": "PROCESSING"},
)
resp = bedrock.invoke_model(
modelId="anthropic.claude-3-haiku-20240307-v1:0",
body=json.dumps({
"anthropic_version": "bedrock-2023-05-31",
"max_tokens": 1024,
"messages": [{"role": "user", "content": msg["prompt"]}],
}),
contentType="application/json", accept="application/json",
)
result = json.loads(resp["body"].read())["content"][0]["text"]
table.update_item(
Key={"job_id": job_id},
UpdateExpression="SET #s = :s, #r = :r, completed_at = :t",
ExpressionAttributeNames={"#s": "status", "#r": "result"},
ExpressionAttributeValues={":s": "COMPLETE", ":r": result,
":t": int(time.time())},
)

# --- POLL HANDLER (API Gateway GET /jobs/{id}) ---
def poll_handler(event, context):
job_id = event["pathParameters"]["id"]
item = table.get_item(Key={"job_id": job_id}).get("Item")
if not item:
return {"statusCode": 404, "body": json.dumps({"error": "job not found"})}
return {"statusCode": 200, "body": json.dumps({
"job_id": item["job_id"],
"status": item["status"],
"result": item.get("result", None),
})}
```

## Pattern B — Async Submit + Webhook Callback

Best for: server-to-server integrations, Zapier/n8n, CI/CD pipelines

**Flow:** Client POSTs with a *callback_url* → Worker processes → Worker POSTs the result to callback_url when done. No polling required — eliminates the N*polling_interval latency overhead.

```
# Worker sends result to client's callback URL
import requests

def worker_handler(event, context):
for record in event["Records"]:
msg = json.loads(record["body"])
callback_url = msg.get("callback_url")
# ... invoke LLM, get result ...
result = "Generated content here"

if callback_url:
requests.post(callback_url, json={
"job_id": msg["job_id"],
"status": "COMPLETE",
"result": result,
}, headers={"X-Signature": compute_hmac(result, SECRET_KEY)},
timeout=10)
```

### ■ Azure & GCP Async Equivalents

Azure: Use Azure Service Bus (queue) + Azure Functions (SBT trigger) + Cosmos DB (state store). Return 202 from Azure API Management immediately. GCP: Use Cloud Tasks (queue) + Cloud Run (worker) + Firestore (state store). Return 202 from Cloud Endpoints or API Gateway immediately.

# 03 Scalability Patterns

## Horizontal Scaling with Queues

The most reliable way to scale an AI workload is to decouple producers from consumers with a durable queue. Workers auto-scale based on queue depth — zero workers when idle, hundreds during peak load.

| | |
|---|---|
| **Queue depth scaling** | Configure auto-scaling triggers: scale out when queue depth > 100 messages, scale in when < 5. AWS: SQS + Lambda (automatic). Azure: Service Bus + KEDA. GCP: Cloud Tasks + Cloud Run. |
| **Concurrency limits** | Set a maximum concurrency on your workers equal to your LLM API quota / average tokens per request. Exceeding this wastes Lambda invocations on 429 errors. |
| **Priority queues** | Use separate queues for interactive (high-priority) vs. batch (low-priority) workloads. Route premium users to a dedicated high-throughput queue with dedicated LLM capacity. |
| **Back-pressure** | If your queue grows unbounded, add a back-pressure mechanism: HTTP 503 to the client when queue depth > threshold, directing them to retry after N seconds. |

## Prompt Caching

Prompt caching stores the KV-cache of a long, static prompt prefix so subsequent requests with the same prefix skip re-processing those tokens. This is one of the highest-ROI optimisations for RAG and agent system prompts.

| Platform | Feature | Cache Duration | Discount | Min Cacheable Tokens |
|---|---|---|---|---|
| AWS Bedrock (Claude) | Prompt caching (beta) | 5 minutes | ~90% on cache hits | 1,024 tokens |
| Azure OpenAI | Prompt caching (automatic) | 1 hour | ~50% on cached | 1,024 tokens |
| GCP Vertex AI | Context caching | Up to 1hr | ~75% on cached | 32,768 tokens |

```
# AWS Bedrock — enable prompt caching with cache_control breakpoints
import boto3, json

bedrock = boto3.client("bedrock-runtime", region_name="us-east-1")
```

```
LONG_SYSTEM_PROMPT = "You are an expert financial analyst..." + " context " * 800 # ~1,200
tokens

response = bedrock.invoke_model(
modelId="anthropic.claude-3-haiku-20240307-v1:0",
body=json.dumps({
"anthropic_version": "bedrock-2023-05-31",
"max_tokens": 512,
"system": [
{
"type": "text",
"text": LONG_SYSTEM_PROMPT,
"cache_control": {"type": "ephemeral"}, # mark for caching
}
],
"messages": [{"role": "user", "content": "Analyse Q3 performance"}],
}),
contentType="application/json", accept="application/json",
)
result = json.loads(response["body"].read())
# Check cache usage in response headers / usage block
usage = result.get("usage", {})
print(f"Cache read tokens: {usage.get('cache_read_input_tokens', 0)}")
print(f"Cache write tokens: {usage.get('cache_creation_input_tokens', 0)}")
```

## Batch Inference

For non-real-time workloads (nightly reports, bulk document processing, dataset annotation), batch inference is 40–60% cheaper than on-demand. All three platforms support it.

| Platform | Service | Input Format | Discount | Max Job Size |
|---|---|---|---|---|
| AWS Bedrock | Bedrock Batch Inference | JSONL in S3 | 50% off on-demand | 50,000 records |
| Azure OpenAI | Batch API | JSONL in Blob Storage | 50% off on-demand | 20M tokens/job |
| GCP Vertex AI | Batch Prediction | JSONL in GCS/BigQuery | ~40% off on-demand | No hard limit |

# 04 Multi-Cloud & Failover Design

## Why Multi-Cloud AI?

| | |
|---|---|
| **Resilience** | A single cloud provider's model endpoint can have outages. In 2024, major LLM APIs experienced multiple multi-hour incidents. Multi-cloud gives you automatic failover. |
| **Model diversity** | No single provider has the best model for every task. Route reasoning tasks to Claude, coding to GPT-4o, and large-context to Gemini 1.5 Pro for optimal quality per use case. |
| **Cost arbitrage** | Model pricing changes frequently. A router can dynamically choose the cheapest provider for a given token budget at any moment. |
| **Compliance** | Some data must stay in specific regions or with specific providers (EU data on Azure EU regions, US gov data on AWS GovCloud). A router handles this transparently. |
| **Avoid lock-in** | Abstracting over providers means you can migrate to a new model in hours, not weeks, when a better option launches. |

## Active-Active vs. Active-Passive

| Strategy | How It Works | Latency | Cost | Best For |
|---|---|---|---|---|
| **Active-Active** | All providers receive traffic simultaneously; router splits by task type or load | Optimal | Full cost on all | High-volume, model-diverse workloads |
| **Active-Passive** | Primary provider handles all traffic; secondary activates only on failure detection | Failover ~5s | Primary only | Cost-sensitive, moderate volume |
| Weighted Round Robin | Traffic split by weight (e.g. 70% AWS, 20% Azure, 10% GCP) | Near-optimal | Proportional | Load distribution + gradual migration |
| Latency-based Routing | Route to whichever provider responds fastest for current region | Best P50 | Variable | Global apps, latency-sensitive chat |

## Abstraction Layer + Failover Code

The key to multi-cloud AI is a thin abstraction layer that normalises the different SDK interfaces into a single **generate(prompt)** call. The router handles provider selection, retry, and failover transparently.

```
import boto3, json, time
from openai import AzureOpenAI
import vertexai
from vertexai.generative_models import GenerativeModel

class MultiCloudLLMRouter:
"""Route LLM calls across AWS Bedrock, Azure OpenAI, and GCP Vertex AI
with automatic failover and exponential backoff retry."""

PROVIDERS = ["bedrock", "azure", "vertex"]

def __init__(self):
self.bedrock = boto3.client("bedrock-runtime", region_name="us-east-1")
self.azure = AzureOpenAI(
api_key=os.environ["AZURE_OPENAI_KEY"],
api_version="2024-02-01",
azure_endpoint=os.environ["AZURE_OPENAI_ENDPOINT"],
)
vertexai.init(project=os.environ["GCP_PROJECT"], location="us-central1")
self.vertex = GenerativeModel("gemini-1.5-flash")
self.health = {p: True for p in self.PROVIDERS}
self.failures = {p: 0 for p in self.PROVIDERS}

def _invoke_bedrock(self, prompt: str, max_tokens: int) -> str:
resp = self.bedrock.invoke_model(
modelId="anthropic.claude-3-haiku-20240307-v1:0",
body=json.dumps({
"anthropic_version": "bedrock-2023-05-31",
"max_tokens": max_tokens,
"messages": [{"role": "user", "content": prompt}],
}),
contentType="application/json", accept="application/json",
)
return json.loads(resp["body"].read())["content"][0]["text"]

def _invoke_azure(self, prompt: str, max_tokens: int) -> str:
resp = self.azure.chat.completions.create(
model="gpt-4o",
messages=[{"role": "user", "content": prompt}],
max_tokens=max_tokens,
)
return resp.choices[0].message.content

def _invoke_vertex(self, prompt: str, max_tokens: int) -> str:
resp = self.vertex.generate_content(prompt)
return resp.text

def generate(self, prompt: str, max_tokens: int = 512,
```

```
    preferred: str = "bedrock") -> dict:
    ordered = [preferred] + [p for p in self.PROVIDERS if p != preferred]
    for provider in ordered:
        if not self.health[provider]:
            continue
        for attempt in range(3):
            try:
                invoke = getattr(self, f"_invoke_{provider}")
                result = invoke(prompt, max_tokens)
                self.failures[provider] = 0  # reset on success
                return {"result": result, "provider": provider}
            except Exception as e:
                wait = (2 ** attempt) + (0.1 * attempt)
                if "429" in str(e) or "throttl" in str(e).lower():
                    time.sleep(wait)
                else:
                    self.failures[provider] += 1
                    if self.failures[provider] >= 3:
                        self.health[provider] = False  # circuit open
                    break
    raise RuntimeError("All providers failed or circuit open")

# Usage
router = MultiCloudLLMRouter()
resp = router.generate("Summarise the Q3 earnings report", preferred="bedrock")
print(f"[{resp['provider']}] {resp['result']}")
```

05

# API Gateway Design

## Request Routing & Versioning

| | |
|---|---|
| **URL versioning** | Always version your AI API: /v1/generate, /v2/generate. Models and response schemas change; versioning lets you migrate clients gradually without breaking existing integrations. |
| **Model routing headers** | Accept an X-Model-Preference: fast\|quality\|cheap header. Route fast to Haiku/GPT-4o-mini/Flash, quality to Sonnet/GPT-4o/Pro, cheap to the current lowest-cost option. |
| **Tenant routing** | Route enterprise tenants to dedicated throughput (PTU / Provisioned Concurrency) and free-tier users to on-demand. Use API key scopes or JWT claims to determine tier. |
| **Canary deployments** | Route 5% of traffic to a new model version. Compare output quality metrics. Ramp to 100% if KPIs improve, roll back if they degrade. |

## Rate Limiting Tiers

| Tier | RPM | TPM | Concurrent Streams | Model Access |
|---|---|---|---|---|
| Free | 10 | 10,000 | 1 | Small models only |
| Developer | 60 | 100,000 | 5 | Small + medium |
| Professional | 300 | 500,000 | 20 | All models |
| Enterprise | Custom | Custom | 100+ | All + dedicated capacity |

## Auth Patterns

| Pattern | Use When | Implementation |
|---|---|---|
| API Key | Simple developer access, CLI tools | Static key in header; rotate every 90 days; scope per tenant in DynamoDB |
| JWT / OAuth 2.0 | User-facing apps, mobile clients | Issue short-lived JWTs (15 min); verify in Lambda authoriser or APIM policy |
| mTLS | Service-to-service, zero-trust | Client cert pinned; AWS ACM, Azure Key Vault, or GCP Certificate Manager |
| IAM Role (SigV4) | Internal AWS services, Lambda-to-Bedrock | No credentials in code; assume role via EC2/ECS/Lambda execution role |

# 06 SLA Engineering

## Latency Targets & P99 Budgets

Latency in AI systems is non-Gaussian — the tail (P99) is often 5-10x the median. Always set SLAs on P95 and P99, never on average. Define budgets per component.

| Component | P50 Target | P95 Target | P99 Target | Key Levers |
|---|---|---|---|---|
| API Gateway | < 10ms | < 50ms | < 100ms | Edge caching, regional deployment |
| Auth / rate limit | < 5ms | < 20ms | < 50ms | DynamoDB DAX, Redis token cache |
| Embedding (RAG) | < 100ms | < 300ms | < 600ms | Batch embed, async retrieval |
| Vector search | < 50ms | < 150ms | < 300ms | ANN index tuning, replicas |
| LLM (first token) | < 500ms | < 1.5s | < 3s | Streaming, provisioned throughput |
| LLM (full response) | < 5s | < 15s | < 30s | Token limits, smaller model for drafts |
| End-to-end (chat) | < 1s TTFT | < 3s | < 5s | Streaming + all above combined |

## Circuit Breaker Pattern

A circuit breaker prevents cascading failures by stopping requests to a failing downstream service and failing fast with a fallback response. Essential for AI systems where a slow LLM endpoint can queue-freeze your entire application.

```
import time, threading
from enum import Enum

class State(Enum):
CLOSED = "closed" # normal — requests pass through
OPEN = "open" # failing — requests fail fast
HALF_OPEN= "half_open" # testing — 1 probe request allowed

class CircuitBreaker:
def __init__(self, failure_threshold=5, recovery_timeout=60, half_open_max=1):
self.state = State.CLOSED
self.failures = 0
self.failure_threshold = failure_threshold
self.recovery_timeout = recovery_timeout
self.half_open_max = half_open_max
self.last_failure_time = None
self._lock = threading.Lock()

def call(self, fn, *args, fallback=None, **kwargs):
```

```
        with self._lock:
            if self.state == State.OPEN:
                elapsed = time.time() - self.last_failure_time
                if elapsed >= self.recovery_timeout:
                    self.state = State.HALF_OPEN
                else:
                    return fallback() if fallback else None

        try:
            result = fn(*args, **kwargs)
            with self._lock:
                self.failures = 0
                self.state = State.CLOSED
            return result
        except Exception as e:
            with self._lock:
                self.failures += 1
                self.last_failure_time = time.time()
                if self.failures >= self.failure_threshold:
                    self.state = State.OPEN
            if fallback:
                return fallback()
            raise

# Usage with Bedrock
cb = CircuitBreaker(failure_threshold=5, recovery_timeout=60)

def cached_fallback():
    return "I'm experiencing high load. Please try again in a moment."

def invoke_llm(prompt):
    # your bedrock/azure/vertex call here
    return bedrock_client.invoke_model(...)

result = cb.call(invoke_llm, "Explain quantum computing",
                 fallback=cached_fallback)
```

## Availability Calculations

| SLA | Downtime / Month | Downtime / Year | Achievable With |
|---|---|---|---|
| 99.0% | 7h 18m | 3d 15h | Single provider, no redundancy |
| 99.5% | 3h 39m | 1d 19h | Single provider + retry logic |
| 99.9% | 43m 49s | 8h 45m | Active-passive multi-cloud failover |
| 99.95% | 21m 54s | 4h 22m | Active-active + circuit breaker |

| | | | |
|---|---|---|---|
| 99.99% | 4m 22s | 52m 35s | Active-active + global load balancing |

# 07 Cost Optimisation Playbook

## Cost Driver Breakdown

In a typical AI workload, token cost dominates but infrastructure is the silent multiplier. Understanding your cost breakdown is the prerequisite to optimising it.

| Cost Driver | Typical % of Bill | Primary Lever |
|---|---|---|
| LLM input tokens | 35–50% | Prompt compression, caching, shorter system prompts |
| LLM output tokens | 25–40% | max_tokens limits, streaming with early stop |
| Embedding generation | 5–10% | Cache embeddings, batch embed, smaller model |
| Vector DB storage | 5–10% | TTL on old chunks, tiered storage |
| Lambda / Function | 2–5% | Right-size memory, provisioned concurrency |
| Data transfer | 2–5% | Same-region inference, VPC endpoints, compression |
| Queue / storage | 1–3% | Message TTL, lifecycle policies on S3/Blob/GCS |

## Optimisation Matrix

| Technique | Effort | Savings | Notes |
|---|---|---|---|
| Model right-sizing | Low | 60–80% | Use Haiku/Flash for classification, formatting, extraction |
| Prompt caching | Low | 50–90% | Most impactful for RAG + agent system prompts over 1K tokens |
| max_tokens limiting | Low | 20–40% | Set realistic per-task limits; reduces both cost and latency |
| Batch inference | Low | 40–60% | For any non-real-time workload; submit JSONL, get results in hrs |
| Response caching | Medium | 30–70% | Cache identical or near-identical prompts in Redis/ElastiCache |
| Prompt compression | Medium | 20–40% | Remove whitespace, truncate examples, use abbreviations in few-shot |
| Streaming + early stop | Medium | 15–30% | Stop generation when answer detected; saves trailing output tokens |
| Provisioned throughput | High | 40–60% | Only viable above a sustained token volume threshold; amortises quickly |

## IaC Cost Controls

- Tag every AI resource with **cost-center**, **environment**, and **team** tags from day one. Without tags, cost attribution is guesswork.
- Set **AWS Budgets / Azure Cost Alerts / GCP Budget Alerts** at 50%, 80%, and 100% of monthly budget. Alert the engineering team, not just finance.
- Use **AWS Cost Anomaly Detection** to automatically flag LLM spend spikes within hours — a misconfigured loop can run up thousands of dollars before a human notices.
- For Step Functions Express Workflows, set **TimeoutSeconds** at the state machine level. A runaway agent workflow without a timeout will execute indefinitely.
- Enforce **max_tokens** as infrastructure policy via a Lambda authoriser — reject requests that exceed your per-tier token cap before they reach the LLM.
- Review your **CloudWatch / Azure Monitor / Cloud Monitoring** dashboards weekly: tokens per request, cost per user/tenant, cache hit rate, and batch vs. on-demand split.

**■ Architecture Checklist — Before Going to Production**

1. Async pattern implemented for all workflows exceeding 10s. 2. Circuit breaker on every LLM client with fallback response. 3. Prompt caching enabled for system prompts over 1,024 tokens. 4. Rate limiting enforced at API Gateway layer, not just in application code. 5. Multi-cloud failover tested — kill primary provider, verify secondary activates in < 10s. 6. P99 latency measured and SLA documented per endpoint. 7. Cost budgets and anomaly alerts configured before first production traffic. 8. Batch inference scheduled for all non-real-time jobs.

**■ Up Next: Guide #6 — Cloud AI Cheat Sheet**

The final guide in the series: a single-page reference for every platform's model names, API endpoints, SDK commands, pricing units, and a decision flowchart for choosing the right model and platform for any use case.