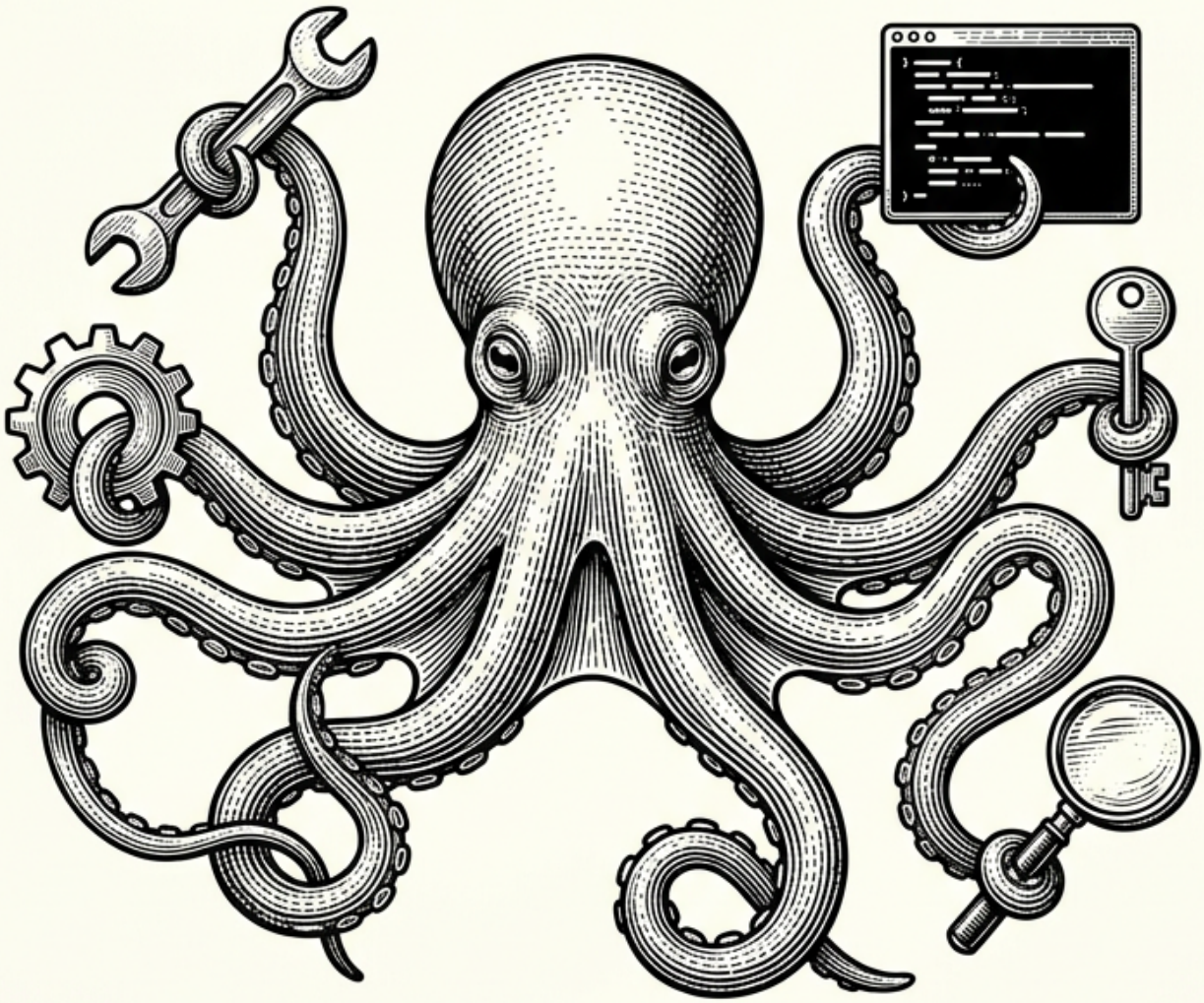
AI Agents in Production

War Stories from a DevOps Engineer

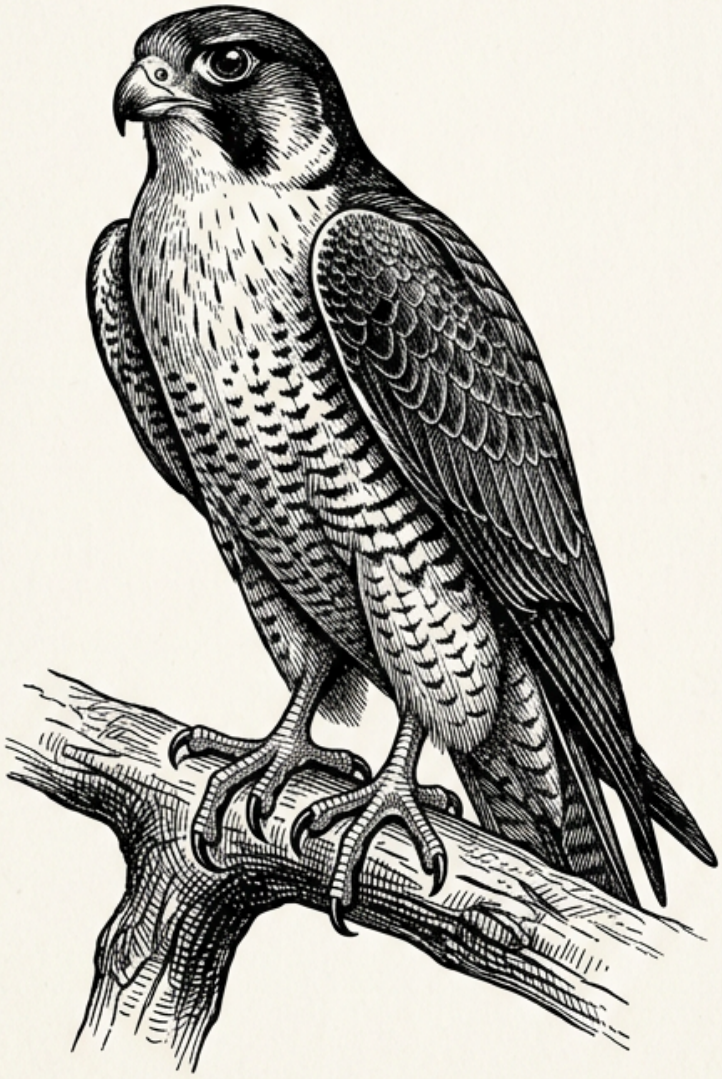


Second Edition

Do Cao Hieu

AI Agents in Production

War Stories from a DevOps Engineer



Second Edition

Do Cao Hieu

AI Agents in Production

War Stories from a DevOps Engineer

Second Edition — March 2026

Contents

AI Agents in Production: War Stories from a DevOps Engineer

Second Edition — March 2026

A practical guide to building, deploying, and operating AI agents in production environments. Based on real experiences running multi-agent systems across 5 servers with Claude, GPT, Gemini, and open-source models.

Table of Contents

Part 5: The Future

- Chapter 15: Scaling Agent Systems
- Chapter 16: Lessons Learned

Appendices

- Appendix A: Tools and Frameworks Reference
- Appendix B: AI Model Pricing Reference
- Appendix C: Production Checklists

PART 1: FOUNDATIONS

Chapter 1: Why This Book Exists

> "The best way to learn something is to get burned by it first." — Every DevOps engineer who survived a $500 cloud bill

---

The Email I Never Expected to Write

It was 2:47 AM on a Tuesday when my phone buzzed. Not a PagerDuty alert, not a server down notification — a billing alert from Google Cloud. The kind that makes your stomach drop before you've even read the number.

$487.32 in 48 hours.

I stared at the screen for a full minute. The Gemini API. A process I had set running three days earlier — what I thought was a contained, well-scoped image generation pipeline for 289 blog posts — had escaped its guardrails and was merrily looping, re-generating, re-processing, burning through tokens like a furnace with no thermostat.

By morning it hit $512. Google's response, after a support ticket and an embarrassingly honest explanation, was a "one-time courtesy credit." Eleven words that simultaneously saved me money and destroyed my pride.

That incident did not start this book. But it crystallized why it needed to exist.

The Gap Between the Demo and Reality

In 2024, every AI vendor showed you the same demo. A developer types a request. An agent thinks for a moment. Code appears. Tests pass. Stars rain from the sky. The crowd applauds.

What the demo never showed: - What happens when the agent loops for 6 hours on a malformed input - How you debug a multi-agent system when three agents disagree - What "100% quota exceeded" looks like when 15 API accounts hit their limits simultaneously - The look on your face when a runaway process burns $500 in 48 hours - Why sometimes a $0.02/M token model makes more sense than a $25/M token model - How you actually monitor, rate-limit, and cost-control autonomous processes in production

This book fills that gap. Not the demo gap — the 3 AM gap.

Who Is Writing This

I run a multi-server infrastructure across three geographic regions: two servers in Singapore, one in Vietnam, backed by Oracle Cloud and fronted by a load balancer. The stack serves real traffic, handles real users, and since mid-2024, runs AI agents as first-class production workloads.

Not "agents" in the sense of a chatbot with memory. Actual autonomous systems:

- **OpenClaw**: A sister AI agent that monitors my Claude Code sessions, handles browser automation, sends Telegram messages, and can provision resources when I'm asleep
- **CLIProxyAPI**: A local routing gateway that manages 15 Antigravity API accounts in round-robin, with automatic failover when any account hits quota
- **Content pipelines**: Batch agents that generate blog posts, images, and metadata across 289 articles
- **Infrastructure agents**: Systems that can SSH into servers, check logs, restart services, and — importantly — make mistakes

I have been running production AI agent infrastructure for eighteen months. I have made almost every mistake this book describes. Some of them I made twice.

My background is DevOps and SRE. I think in uptime SLOs, cost-per-request, and blast radius. I care about things like: what happens when this breaks at 3 AM? Who pays for the mistake? How do I get paged when it goes wrong?

This book is written from that perspective.

What Changed From 2024 to 2026

When I started building agent infrastructure in 2024, the landscape looked like this:

Capability	2024	2026
Best model	Claude 3.5 Sonnet	Claude Opus 4.6
Context window	128K tokens	200K standard, 1M beta
Tool use reliability	~70% success rate	95%+ on well-scoped tasks
Multi-agent frameworks	Experimental	Production-ready
Cost (best model)	$15/MTok input	$5/MTok input (Opus 4.6)
Agent debugging tooling	Nonexistent	Emerging
Quota management	Manual	Partially automatable

The models got dramatically better. Claude 3.5 → 4.6 was not a marketing bump — it was a qualitative leap in reasoning reliability, tool use accuracy, and long-context coherence. GPT-4 → 5.x brought similar improvements. Gemini went from a promising but inconsistent model to something genuinely competitive on price-performance.

But here is what did not change: **production AI agents still break in all the ways production software breaks**, plus a whole new category of AI-specific failure modes that nobody had names for in 2024.

The tooling for observability, cost control, and multi-agent coordination has improved, but it is still at the "early 2015 Kubernetes" stage. You can make it work. It requires craft.

The Context Window Revolution

One specific change deserves attention because it reshaped everything about how agents work.

In 2024, 128K tokens felt large. In practice, once you added a system prompt, conversation history, tool definitions, and a few rounds of tool outputs, you were at 80K and sweating. Context pressure forced constant trade-offs: summarize more aggressively, prune history earlier, break tasks into smaller chunks.

In 2026, 1M context windows changed the equation. You can now feed an agent an entire codebase, complete conversation history, and dense tool output without hitting limits. This enables:

- **Longer autonomous runs**: Agents can work on complex, multi-step tasks without losing context
- **Richer memory**: Full conversation history instead of compressed summaries
- **Better debugging**: Agents can review their own prior decisions before acting
- **Cross-session continuity**: Context files that span days of work

The trade-off is cost. A 1M token context costs money every time you send a request. Context engineering — the craft of deciding what goes in the window and what stays out — became one of the most important skills in agent development. We'll cover it extensively in Chapter 7.

The Foundational War Story: 15 Accounts Suspended in One Day

Before we go further, let me tell you about the day that fundamentally changed how I think about AI infrastructure.

Background

I run AI workloads through a local proxy service called CLIProxyAPI. This proxy sits between my agents and the model providers, handling authentication, routing, rate limiting, and failover. To extend my capacity, I had registered 15 accounts with a service called Antigravity (an API access aggregator that provides free/subsidized Gemini API credits).

The system was elegant: agents hit the local proxy, the proxy round-robins across 15 accounts, and when any account hits its quota limit, the proxy automatically switches to the next available account. 15 accounts meant 15x the effective rate limit. Smart, right?

What Happened

On a Thursday morning in February 2026, I launched a large batch content generation job: 289 blog posts, each requiring title generation, outline creation, content generation, and image metadata generation. Four API calls per post. 1,156 total API calls.

The first indication something was wrong came at 11:23 AM: my monitoring dashboard showed quota exhausted on Account 1. Normal. The proxy should switch to Account 2.

It did. Account 2 exhausted by 11:31 AM.

By noon, all 15 accounts were at 100% quota. The proxy had no more accounts to fail over to. But here was the real problem: I had not noticed. The agents kept trying, the proxy kept returning errors, and my batch job — designed to retry on failure — was hammering the endpoint, burning the few remaining tokens across all 15 accounts trying to get *any* response.

By 2 PM, all 15 Antigravity accounts were suspended.

The cause: The batch job was far more token-heavy than I had estimated. I had not accounted for the system prompts (which I had expanded significantly two days earlier), the output token counts (Gemini generates verbose JSON), or the fact that image metadata generation alone was consuming 3x what I expected.

The Cascade

The suspension of all 15 accounts did not just kill the batch job. It killed everything that ran through those accounts:

- OpenClaw's health checks (which were running on a 3-minute cron, consuming tokens)
- Two background agent processes monitoring my servers
- A content pipeline for another project

I spent six hours on Thursday afternoon doing triage. Account appeals to Antigravity (most were restored within 24 hours). Manually completing the batch job through a different API key. Rewriting the health check cron to use a cheaper model.

What Changed After

This incident forced a set of decisions I should have made months earlier:

1. Token budget tracking per job

Before any batch job now runs, I estimate token consumption per request:

```
# Rough estimate before launching a batch
estimated_input_tokens = (
    system_prompt_tokens +      # Measure this once, cache it
    example_tokens +            # Few-shot examples if any
    request_content_tokens      # Per-item content
)
estimated_output_tokens = expected_output_length
estimated_cost = (
    estimated_input_tokens * input_price_per_token +
    estimated_output_tokens * output_price_per_token
) * batch_size

print(f"Estimated cost: ${estimated_cost:.2f}")
print(f"Estimated tokens: {(estimated_input_tokens + estimated_output_tokens) * batch_size:,}")
```

If the estimate is above a threshold, I require explicit confirmation before proceeding.

2. Circuit breakers on retry logic

The batch job retried on *every* failure, including quota exhaustion. This was the direct cause of the cascade. After this incident, every batch job has a circuit breaker:

```python
class QuotaCircuitBreaker:
    def __init__(self, threshold: int = 5, window_seconds: int = 60):
        self.failures = []
        self.threshold = threshold
        self.window_seconds = window_seconds
        self.open = False

    def record_failure(self, error_type: str):
        now = time.time()
        self.failures = [f for f in self.failures
                         if now - f['time'] < self.window_seconds]
        self.failures.append({'time': now, 'type': error_type})

        if len(self.failures) >= self.threshold:
            self.open = True
            raise CircuitOpenException(
                f"Circuit breaker opened after {self.threshold} failures. "
                f"Pausing to prevent quota cascade."
            )
```

3. Health checks do not use expensive models

Every health check that was burning tokens was using the same model as the main workload. After the incident, health checks use the cheapest available model (Gemini Flash, or for non-AI checks, no model at all). I also deleted all cron-based health check jobs and replaced them with event-driven checks that only run when there is actual traffic.

4. Account segmentation

The 15 Antigravity accounts are now split into pools: 10 for production workloads, 5 reserved for health checks and monitoring. Batch jobs cannot drain the monitoring pool.

5. Context file optimization

As part of the emergency response, I audited every context file that agents were loading. The results were startling:

Context File	Before	After	Reduction
System prompt	4,200 tokens	380 tokens	91%
Tool definitions	2,800 tokens	690 tokens	75%
Memory context	2,030 tokens	127 tokens	94%
Total	**9,030 tokens**	**797 tokens**	**91%**

Every agent request was consuming 9,030 tokens *before any actual content.* At scale, this is devastating. The optimization work — stripping verbose documentation from system prompts, compressing tool definitions, moving memory to on-demand retrieval — reduced per-request baseline costs by 91%.

We will cover context engineering in detail in Chapter 7. The short version: your system prompt is not a README file. Write it like it costs money, because it does.

What This Book Covers

This book is organized around the practical lifecycle of AI agents in production. Part 1 covers foundations — what agents actually are, how to choose models, and how to think about costs. Part 2 covers architecture — how to design agent systems that are observable, reliable, and recoverable. Part 3 covers the hard parts — multi-agent coordination, memory systems, and context engineering. Part 4 covers operations — monitoring, incident response, cost control, and security.

Each chapter includes at minimum one war story from my actual infrastructure. These are not hypotheticals. The mistakes described are mistakes I made, with real costs (financial and otherwise). The solutions described are solutions currently running in my production environment.

Code examples are real. Configs are real (with credentials redacted). Pricing numbers are current as of March 2026, though the models change frequently enough that you should verify before making major architectural decisions.

What This Book Does Not Cover

This is not a book about: - Building your first chatbot - Prompt engineering for consumer use cases - How to use Claude or ChatGPT from the web interface - Machine learning theory or model training - Building custom models

If you are new to AI APIs entirely, you may want to start with a more introductory resource. This book assumes you have at least used an AI API, understand basic HTTP concepts, and are comfortable with a terminal.

Who This Book Is For

Primary audience: DevOps engineers, SREs, and platform engineers who are being asked to run AI agent workloads in production. People who have inherited an agent system someone else built and are wondering why it keeps breaking. People who are about to build one and want to avoid the obvious mistakes.

Secondary audience: Backend engineers building agent-powered features who need to understand infrastructure concerns. Tech leads making architectural decisions about AI integration. Engineering managers who want to understand what their teams are telling them about AI agent complexity.

Not for: Researchers, academics, or people primarily interested in the theoretical aspects of AI. The orientation here is entirely practical: what works, what breaks, what it costs, and how to fix it.

A Note on Honesty

I will be direct about failures throughout this book. The $500 Gemini bill is not the only embarrassing incident you will read about. There are chapters describing multi-agent systems that deadlocked, agents that modified the wrong config and broke production services, and a security incident involving a token that leaked through chat history.

This is intentional. The DevOps community has a strong tradition of blameless postmortems and honest incident reports because we learned — often painfully — that pretending failures do not happen prevents learning. I am applying that tradition to AI agent infrastructure.

The AI industry, by contrast, has a strong tradition of only showing demos that work. This book is the other side of that.

How to Use This Book

If you are evaluating AI agents for production use: Read Chapters 1-3 (Part 1) and Chapter 12 (Cost Control). These give you the framing and the numbers you need to make the evaluation.

If you are designing an agent system: Read Chapters 4-8 (Parts 1-2). The architecture chapters will save you from the most common structural mistakes.

If you are debugging an existing agent system: Skip to Chapter 14 (Debugging War Stories). Then read whatever background chapters address your specific issue.

If you are getting paged about an agent in production: Chapter 13 (Incident Response for AI Systems) first. Then the rest can wait until morning.

If you want to understand the whole picture: Read it in order. There are forward references, but each chapter is designed to stand alone if needed.

The Setup That Makes This All Real

Throughout this book, I'll reference my actual infrastructure. Here is the quick map so you can orient yourself when I mention specific components:

```
Infrastructure Map (March 2026)
================================

Internet → Load Balancer (Oracle)
                ↓


  SG-01                SG-02
(Singapore)          (Singapore)
Primary web          Agent runner
Traffic + API        Background jobs
```

```
                    VN-01
                  (Vietnam)
                Data processing
                Content pipeline

Local workstation (me):
     CLIProxyAPI (port 3000)
           Antigravity Pool A (10 accounts) → Gemini API
           Antigravity Pool B (5 accounts)  → Monitoring only
           Direct API keys (Claude, OpenAI)

AI Agents running:
     Claude Code (primary development assistant)
     OpenClaw (sister agent, browser + messaging)
     Content Pipeline (batch generation)
     Infrastructure Monitor (event-driven, not cron)
```

When I say "the agent SSH'd into SG-01," this is what I mean. When I say "the proxy rotated to the next Antigravity account," this is the system doing the rotating.

Before We Begin

I want to be clear about something before we go further: AI agents in production are genuinely powerful. The reason I run this infrastructure is not masochism. OpenClaw has saved me hours every week. My content pipeline produces work in hours that would take days manually. Claude Code, running as an agent with filesystem access, has written, tested, and deployed features while I slept.

The premise of this book is not "AI agents are dangerous, avoid them." The premise is "AI agents are powerful tools that require the same professional discipline as any other production infrastructure."

You would not deploy a new database without monitoring, backups, and an incident response plan. You would not run a job queue without dead letter queues, retry limits, and circuit breakers. AI agents deserve the same rigor.

The war stories in this book are not arguments against building with AI agents. They are arguments for building with AI agents *properly*.

Let's get into it.

Key Takeaways

- AI agents in production have a gap between demo promises and operational reality that this book addresses directly
- Real production agent infrastructure requires monitoring, cost control, circuit breakers, and incident response — the same as any production system

- The $500 Gemini incident and the 15 suspended accounts were both caused by missing guardrails, not model failures
- Context engineering (controlling what goes into each request) is one of the highest-leverage optimizations available — 91% token reduction is achievable
- The AI landscape changed significantly from 2024→2026: models improved dramatically, but operational challenges remain
- This book is written from a DevOps/SRE perspective: uptime, cost, blast radius, and 3 AM pagerability

---

Next: Chapter 2 — What Are AI Agents, Really?

Chapter 2: What Are AI Agents, Really?

> "An agent is just a model in a loop with access to tools. The terrifying part is how much that matters." — Senior engineer, post-incident review

---

The Definition Problem

Ask ten people what an "AI agent" is and you will get twelve definitions. Marketing teams use it to mean any AI feature. Researchers use it for systems with formal autonomy properties. Vendors use it for whatever they are selling this quarter.

For the purposes of this book, here is the working definition:

An AI agent is a system that uses a language model to decide what actions to take, then takes those actions, and continues this cycle until a goal is met or a termination condition is reached.

That is it. No magic. No consciousness. No sentience. A model in a loop with tools and a stopping condition.

This definition is deliberately unglamorous, because the engineering challenges of AI agents are not glamorous. They are the same challenges that exist in any autonomous system: state management, error recovery, observability, cost control, and the deeply human question of "what happens when it does something I did not expect?"

Let us build up from first principles.

What a Language Model Actually Is

Before agents, we need to be clear about what language models do, because many agent failures come from misunderstanding this.

A language model is a function: **given a sequence of tokens (text), predict the probability distribution over the next token.**

That is the entire mechanism. Everything else — tool use, reasoning, code generation, instruction following — emerges from training this function on enormous amounts of human-generated text, then fine-tuning it to follow instructions and use provided tools.

This has a critical implication: **language models do not have state between calls.** Every API call is stateless. The model does not remember your previous call. It only sees what you include in the current request's context window.

When we talk about agents with "memory," we are talking about systems that explicitly include relevant past information in each new request. The memory is in the code, not in the model.

This is not a limitation to work around. It is a design principle to internalize. Once you understand that memory is an engineering choice, not a model feature, you start making better decisions about what to remember, how to structure it, and when to discard it.

The Agent Loop

Every AI agent, regardless of framework or complexity, executes a variation of this loop:

```
                AGENT LOOP

1. OBSERVE
   Gather current state:
   - Task description
   - Previous actions + results
   - Available tools
   - Memory/context


2. PLAN
   Call language model with context
   Model decides: continue/act/stop


3. EXECUTE
   If action chosen: run the tool
   Capture result (success or error)


4. EVALUATE
   Did we reach the goal?
   Should we continue?
   Did something go wrong?


   Loop back to OBSERVE
   (or terminate)
```

This loop runs until one of: - The model decides the task is complete - A termination condition is met (step limit, time limit, budget limit) - An unrecoverable error occurs - External signal stops the loop (human in the loop, circuit breaker)

Here is the same loop in Python — a minimal but functional agent implementation:

```
import anthropic
import json
from typing import Any

client = anthropic.Anthropic()

def run_agent(task: str, tools: list[dict], max_steps: int = 20) -> str:
    """
    Minimal agent loop. Runs until completion or step limit.
    Returns final response text.
    """
    messages = [{"role": "user", "content": task}]
    step = 0

    while step < max_steps:
        step += 1
        print(f"[Step {step}] Calling model...")

        # PLAN: Ask the model what to do
        response = client.messages.create(
            model="claude-opus-4-6",
            max_tokens=4096,
            tools=tools,
            messages=messages,
        )

        print(f"[Step {step}] Stop reason: {response.stop_reason}")

        # EVALUATE: Did we finish?
        if response.stop_reason == "end_turn":
            # Model decided it's done
            final_text = next(
                block.text for block in response.content
                if hasattr(block, "text")
            )
            return final_text

        # OBSERVE: Collect tool calls from response
        tool_uses = [
            block for block in response.content
            if block.type == "tool_use"
```

```
        ]

        if not tool_uses:
            # Model stopped but didn't finish and didn't call tools
            # This is an unexpected state - handle it
            raise RuntimeError(f"Agent stuck: stop_reason={response.stop_reason}, no tool calls

        # Add assistant message to history
        messages.append({"role": "assistant", "content": response.content})

        # EXECUTE: Run each tool call
        tool_results = []
        for tool_use in tool_uses:
            print(f"[Step {step}] Executing tool: {tool_use.name}")
            result = execute_tool(tool_use.name, tool_use.input)
            tool_results.append({
                "type": "tool_result",
                "tool_use_id": tool_use.id,
                "content": json.dumps(result),
            })

        # Add results to history - loop continues
        messages.append({"role": "user", "content": tool_results})

    raise RuntimeError(f"Agent exceeded max_steps={max_steps}")


def execute_tool(name: str, inputs: dict[str, Any]) -> Any:
    """Dispatch tool calls to actual implementations."""
    tool_registry = {
        "read_file": tool_read_file,
        "write_file": tool_write_file,
        "run_command": tool_run_command,
        "search_web": tool_search_web,
    }

    if name not in tool_registry:
        return {"error": f"Unknown tool: {name}"}

    try:
        return tool_registry[name](**inputs)
    except Exception as e:
        # Return errors as data - let the model decide what to do
        return {"error": str(e), "tool": name, "inputs": inputs}
```

This is the skeleton of every agent system I run in production, extended with: - Token counting and budget enforcement before each step - Structured logging of every model call and tool execution -

Timeout handling at the tool level - Circuit breakers on repeated failures - Persistent state so runs can be resumed after interruption

We will cover all of these in later chapters. For now, internalize the loop: **Observe → Plan → Execute → Evaluate → repeat**.

What Distinguishes an Agent from a Chatbot

The distinction matters operationally, not philosophically.

Property	Chatbot	Copilot	Agent
Conversation turns	Human-driven	Human-driven	Self-driven
Tool access	Minimal or none	Read-only	Read and write
Side effects	None	Suggestions only	Direct mutations
Duration	Single turn	Single turn	Multi-turn, autonomous
Human in loop	Every turn	Every turn	Optional or none
Cost profile	Per-query	Per-query	Accumulating
Failure blast radius	Low	Low	Potentially high
Observability needs	Low	Low	High

A chatbot waits for you to say something. An agent decides what to do next itself.

A copilot (GitHub Copilot, Claude in an editor) suggests things. You accept or reject. It cannot change files without your approval.

An agent takes actions. It can read files, write files, run shell commands, call APIs, modify databases — whatever tools you give it. If you give an agent write access to your production database and it makes a wrong decision, the database is wrong. There is no "are you sure?" prompt unless you build one.

This is why the engineering discipline around agents matters so much more than around chatbots. **The blast radius of an agent failure scales with the permissions you grant it.**

Types of Agent Architecture

Single Agent

The simplest architecture: one model, one loop, one set of tools.

```
User → Agent (Model + Tools) → Result
```

When to use: Focused tasks with clear scope. A single agent that manages deployments, or summarizes logs, or generates content for a specific pipeline. The agent does one class of thing and does it well.

Failure modes: Single point of failure. If the model gets confused, there is no second opinion. Long tasks accumulate context until the window fills, requiring careful context management.

My usage: Content generation per blog post. The agent receives a title and outline, generates content, writes it to a file. Clean input, clean output, finite scope.

Multi-Agent (Orchestrator + Specialists)

An orchestrator agent breaks down complex tasks and delegates to specialist subagents.

```
User → Orchestrator
              Research Agent → results
              Writing Agent  → draft
              Review Agent   → feedback
              Final output
```

When to use: Tasks that naturally decompose into distinct phases. When you want specialists — a research-optimized agent, a code-specialized agent, a review agent with different criteria.

Failure modes: Orchestrator failures cascade. Communication between agents requires careful interface design (what format does the orchestrator pass to the writer? What does the writer return?). Debugging is harder because errors can originate at any level.

My usage: The Claude Code + OpenClaw pairing. Claude Code handles implementation; OpenClaw handles browser automation, messaging, and tasks that require visual confirmation. They communicate through a shared workspace directory and a messaging system.

Mesh / Swarm

Multiple agents operate in parallel, potentially monitoring each other, with no single orchestrator.

```
Agent A ↔ Agent B

Agent C ↔ Agent D
```

When to use: High-throughput tasks where parallelism matters more than coordination. Monitoring tasks where redundancy is valuable.

Failure modes: Complex. Race conditions. Duplicate work. Coordination overhead that can exceed the benefit of parallelism. Token costs multiply with each agent.

My experience: I ran a mesh monitoring system in late 2025. Six agents, each responsible for a subset of services, reporting into a shared state store. The monitoring worked. The agents also ran health checks every 3 minutes each, burning tokens continuously. The cost was significant enough that I shut it down and replaced it with event-driven monitoring. More on this in the war story section.

The OpenClaw Architecture: A Real Multi-Agent System

Rather than invent a hypothetical, let me describe the actual multi-agent architecture running on my infrastructure as of March 2026.

```
                    OpenClaw System
```

```
 Gateway                    Orchestrator

Receives         - Parses incoming events
events             - Decides which agent to use
via Tele-          - Manages conversation context
gram/HTTP          - Routes results back
```

```
Browser Agent       Research Agent      Comms Agent

Playwright          Web search           Telegram
Screenshot          Summarization        WhatsApp
Form fill           Fact extraction      Discord
Navigation
```

```
                  Shared Memory

~/.openclaw/workspace/
   memory/YYYY-MM-DD.md    (daily logs)
   inbox/from-openclaw.md  (tasks for Claude)
   inbox/from-claudecode.md (tasks for OpenClaw)
   research/               (research outputs)
   MEMORY.md               (long-term memory)
```

```
               Communicates with
```

```
                   Claude Code

                  My primary dev
                  assistant with
                  filesystem and
                  terminal access
```

The key design decisions in this architecture:

1. **Separation of concerns**: Each subagent has a narrow scope. The browser agent does not

know about Telegram. The research agent does not have filesystem write access. Narrow scope limits blast radius and makes debugging tractable.

2. File-based communication: Agents communicate through files in `~/.openclaw/workspace/`, not direct API calls to each other. This means communication is: - Persistent (survives restarts) - Inspectable (I can read the inbox directory to see what was sent) - Decoupled (agents do not need to be running simultaneously) - Debuggable (I can manually write messages to the inbox to test behavior)

3. Explicit memory management: Memory is not just accumulated conversation history. OpenClaw writes structured daily logs, maintains a long-term MEMORY.md file, and passes relevant context to subagents rather than the full history. This keeps context windows manageable.

4. Event-driven, not polling: OpenClaw does not run on a schedule, checking things every N minutes. It responds to events (Telegram messages, Claude Code notifications, system events). This is the lesson from the mesh monitor experiment.

Here is the simplified event handling loop in the orchestrator:

```
class OpenClawOrchestrator:
    def __init__(self):
        self.model = "claude-sonnet-4-6"  # Fast and capable enough for routing
        self.specialist_models = {
            "browser": "claude-opus-4-6",    # Needs best reasoning for web nav
            "research": "claude-sonnet-4-6", # Good balance for structured research
            "comms": "claude-haiku-4-5",     # Simple formatting, no need for Opus
        }
        self.memory = MemoryManager("~/.openclaw/workspace/")

    def handle_event(self, event: dict) -> None:
        """Entry point for all incoming events."""
        # Load relevant context (not all context - just relevant)
        context = self.memory.load_relevant(event)

        # Routing decision - cheap model for routing
        route = self._decide_route(event, context)

        if route.requires_specialist:
            result = self._delegate_to_specialist(
                specialist=route.specialist,
                task=route.task_description,
                context=context,
            )
        else:
            # Orchestrator handles simple tasks directly
            result = self._handle_directly(event, context)

        # Persist outcome
        self.memory.record(event, result)
```

```
        # Notify if needed
        if route.should_notify:
            self._send_notification(result, route.notification_channel)

    def _decide_route(self, event: dict, context: dict) -> Route:
        """
        Use a fast/cheap model just for routing.
        Do NOT use Opus here - it's overkill for classification.
        """
        response = client.messages.create(
            model="claude-haiku-4-5",  # Routing is cheap
            max_tokens=256,
            system=ROUTING_SYSTEM_PROMPT,
            messages=[{
                "role": "user",
                "content": f"Event: {json.dumps(event)}\nContext summary: {context['summary']}"
            }]
        )
        return Route.parse(response.content[0].text)
```

The routing model is always the cheapest option that can do the job reliably. This is a recurring theme: match model capability to task complexity. Routing decisions need classification, not reasoning. Use Haiku for classification. Save Opus for the tasks that actually need it.

Tools: The Agent's Hands

An agent without tools is just a chatbot that thinks harder. Tools are what give agents the ability to affect the world.

Tool design is one of the most important and underrated aspects of agent development. Poorly designed tools lead to: - Agents that call tools with wrong arguments (unclear interfaces) - Unrecoverable errors (non-idempotent tools without safeguards) - Security vulnerabilities (tools with excessive permissions) - Runaway costs (tools that themselves consume tokens or money)

Tool Design Principles

1. Make tools atomic and focused

A tool that does one thing is easier to use correctly than a tool that does many things. Instead of:

```
# Bad: tool does too much
@tool
def manage_files(action: str, path: str, content: str = None, recursive: bool = False):
    """Create, read, update, delete files. action can be 'create', 'read', 'update', 'delete'.
    ...
```

Prefer:

```python
# Good: separate tools for separate concerns
@tool
def read_file(path: str) -> str:
    """Read the contents of a file at the given path."""
    ...

@tool
def write_file(path: str, content: str) -> dict:
    """Write content to a file. Creates if not exists, overwrites if exists."""
    ...

@tool
def delete_file(path: str) -> dict:
    """Delete a file. Returns error if file does not exist."""
    ...
```

The model has to decide which tool to call. Give it simple, unambiguous choices.

2. Return structured, informative results

The tool result is the model's next "observation." Make it rich enough for the model to understand what happened.

```python
# Bad: minimal result
def deploy_service(service: str, version: str) -> str:
    run_deploy(service, version)
    return "done"

# Good: structured result with context
def deploy_service(service: str, version: str) -> dict:
    start_time = time.time()
    try:
        result = run_deploy(service, version)
        return {
            "status": "success",
            "service": service,
            "version": version,
            "duration_seconds": time.time() - start_time,
            "endpoint": result.endpoint,
            "health_check": result.health_status,
        }
    except DeploymentError as e:
        return {
            "status": "error",
            "service": service,
            "version": version,
            "error": str(e),
            "error_type": type(e).__name__,
            "recoverable": e.is_recoverable,
```

```
            "suggested_action": e.suggested_action,
        }
```

3. Build safeguards into dangerous tools

Tools with destructive side effects should include confirmation mechanisms or limits:

```
@tool
def run_shell_command(
    command: str,
    working_directory: str = "/tmp",
    timeout_seconds: int = 30,
    allow_network: bool = False,
) -> dict:
    """
    Run a shell command. Restricted to safe paths by default.
    Will refuse commands that match dangerous patterns.
    """
    DANGEROUS_PATTERNS = [
        r"rm\s+-rf\s+/",
        r":\(\)\{.*\}",  # Fork bomb
        r"dd\s+if=/dev/",
        r"mkfs\.",
    ]

    for pattern in DANGEROUS_PATTERNS:
        if re.search(pattern, command):
            return {
                "status": "refused",
                "reason": f"Command matches dangerous pattern: {pattern}",
                "command": command,
            }

    # Restrict working directory to safe paths
    safe_roots = ["/tmp", "/home/agent", "/workspace"]
    if not any(working_directory.startswith(root) for root in safe_roots):
        return {
            "status": "refused",
            "reason": f"Working directory {working_directory} not in safe paths",
        }

    try:
        result = subprocess.run(
            command,
            shell=True,
            cwd=working_directory,
            capture_output=True,
            text=True,
```

```
            timeout=timeout_seconds,
        )
        return {
            "status": "success",
            "stdout": result.stdout[-4000:],  # Truncate long outputs
            "stderr": result.stderr[-1000:],
            "return_code": result.returncode,
        }
    except subprocess.TimeoutExpired:
        return {"status": "timeout", "command": command, "timeout": timeout_seconds}
```

4. Make tools observable

Every tool call should be logged. You need to know what your agent did.

```
import functools
import logging

def observable_tool(func):
    """Decorator that logs all tool calls."""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        call_id = str(uuid.uuid4())[:8]
        logger.info(f"TOOL_CALL [{call_id}] {func.__name__}: args={args} kwargs={kwargs}")
        start = time.time()
        try:
            result = func(*args, **kwargs)
            duration = time.time() - start
            logger.info(f"TOOL_RESULT [{call_id}] {func.__name__}: duration={duration:.2f}s sta
            return result
        except Exception as e:
            duration = time.time() - start
            logger.error(f"TOOL_ERROR [{call_id}] {func.__name__}: duration={duration:.2f}s err
            raise
    return wrapper
```

War Story: The Agent Mesh Monitor That Burned Itself Down

In Q4 2025, I built what seemed like an elegant solution to infrastructure monitoring.

The Setup

Six lightweight agents, each assigned to monitor a subset of services: - Agent Alpha: Web servers (SG-01, SG-02) - Agent Beta: Database (PostgreSQL + Redis) - Agent Gamma: API services (CLIProxyAPI, application endpoints) - Agent Delta: Content delivery and static assets - Agent Epsilon: SSL certificates and DNS - Agent Zeta: AI workload queue depth

Each agent ran on a 3-minute cron schedule, checking its assigned services and writing status to a shared state file. A seventh "coordinator" agent read the state file every 10 minutes and sent a daily summary to Telegram.

The design was genuinely solid from a reliability standpoint. Agents were independent — a failure in Agent Alpha did not affect Agent Beta. The shared state was a simple JSON file with file locking to prevent concurrent writes. No agent had write access to anything except the state file.

The Problem

Token consumption at scale.

Each agent call, including system prompt, tool definitions, and the monitoring check itself, consumed approximately:

- System prompt: 380 tokens
- Tool definitions (6 monitoring tools): 890 tokens
- Task description: ~120 tokens
- Response: ~200-400 tokens
- Tool execution + result: ~150-300 tokens

Total per check: roughly **1,700 tokens per agent run**.

Six agents, every 3 minutes, equals 2 agent runs per minute across the fleet, 2,880 runs per day. At 1,700 tokens per run:

```
2,880 runs/day × 1,700 tokens/run = 4,896,000 tokens/day
At $3/MTok input + $15/MTok output (Sonnet 4.6):
~$0.012 per run × 2,880 runs = $34.56/day
```

The coordinator agent added another ~$3/day.

$37.56 per day to know if my servers were up.

For context: my actual server costs are $48/month. I was spending more on monitoring than on compute.

The Worse Problem

The agents detected issues. I had built that part correctly. What I had not anticipated was that *the agents detecting issues also generated more tokens.* A normal check: 1,700 tokens. A check that found a problem and needed to investigate further: 4,000-8,000 tokens, because the agent would call additional diagnostic tools, look at logs, cross-reference with other service states.

On a bad infrastructure day — say, a certificate about to expire, two services with elevated error rates, and a database connection pool running hot — the monitoring agents would each do deeper dives, generating 5x the normal token consumption. The days when I most needed monitoring were the days monitoring cost the most.

The Decision

After three weeks, I shut down the entire mesh monitoring system and replaced it with:

1. Push-based alerting: Services push alerts to a webhook when they detect their own problems. No polling. No agent involvement until there is an alert worth investigating.

2. Single coordinator agent, event-driven: The coordinator runs only when an alert arrives. It receives the alert, calls relevant diagnostic tools, and sends a structured summary. One agent, on demand, instead of six agents on a schedule.

3. Simple HTTP health checks: A lightweight script (no AI, no model calls) polls `/health` endpoints every 30 seconds and fires alerts on failures. 0 tokens consumed. A cron job that runs a `curl` command costs nothing.

4. AI agent for diagnosis, not detection: When an alert fires, *then* I bring in the AI agent to help diagnose. The agent examines logs, cross-references services, and suggests remediation. This is a good use of agent capability — complex reasoning about ambiguous data. Detection is a bad use — it is pattern matching on known states, better done by purpose-built monitoring tools.

The revised system costs approximately $0.50/day in AI tokens — for the occasional diagnostic runs when something actually goes wrong.

The Lesson

Use AI agents for tasks that require intelligence. Use purpose-built tools for tasks that require reliability and repeatability.

Health checks are not intelligent tasks. Checking if an HTTP endpoint returns 200 is not something that benefits from a language model. It benefits from a simple, reliable, cheap program that does exactly that and nothing else.

AI agents shine when: - The task has ambiguous inputs that require interpretation - The path to completion is not known in advance - Multiple sources of information need to be synthesized - The problem requires reasoning under uncertainty

AI agents are expensive and overkill when: - The task is well-defined and deterministic - The same logic runs on a schedule regardless of context - Speed and cost matter more than intelligence - Simple pattern matching suffices

The mesh monitor was an expensive way to learn this lesson. I will not make that mistake again — and I am passing the lesson to you early so you do not have to.

Agent vs. Chatbot vs. Copilot: The Practical Table

Here is the comparison I actually use when deciding which category of tool to reach for:

Question	Chatbot	Copilot	Agent
Does it need to take actions?	No	Suggestions only	Yes
Does it need to run autonomously?	No	No	Yes
Is there a human reviewing each step?	Yes	Yes	Optional

Question	Chatbot	Copilot	Agent
Does it have write access to systems?	No	Rarely	Often
Will it run while I'm asleep?	No	No	Yes
What happens when it makes a mistake?	Rephrase and retry	Reject suggestion	Potentially bad things
Do I need audit logs?	Nice to have	Nice to have	Mandatory
Do I need cost controls?	Minimal	Minimal	Critical

If you are building something and the answers in the "Agent" column scare you, good. That fear is appropriate and should motivate proper engineering. If the answers in the "Agent" column do not scare you, re-read the question about "what happens when it makes a mistake."

The Node.js Version: For the Server-Side Engineers

For teams building agents in Node.js/TypeScript, here is the equivalent agent loop:

```typescript
import Anthropic from "@anthropic-ai/sdk";

const client = new Anthropic();

interface ToolResult {
  tool_use_id: string;
  content: string;
  type: "tool_result";
}

interface AgentOptions {
  task: string;
  tools: Anthropic.Tool[];
  maxSteps?: number;
  onStep?: (step: number, action: string) => void;
}

async function runAgent(options: AgentOptions): Promise<string> {
  const { task, tools, maxSteps = 20, onStep } = options;

  const messages: Anthropic.MessageParam[] = [
    { role: "user", content: task },
  ];

  for (let step = 1; step <= maxSteps; step++) {
    onStep?.(step, "calling_model");
```

```
  const response = await client.messages.create({
    model: "claude-opus-4-6",
    max_tokens: 4096,
    tools,
    messages,
  });

  // Agent finished
  if (response.stop_reason === "end_turn") {
    const textBlock = response.content.find((b) => b.type === "text");
    if (!textBlock || textBlock.type !== "text") {
      throw new Error("Agent finished without text response");
    }
    return textBlock.text;
  }

  // Collect tool calls
  const toolUseBlocks = response.content.filter(
    (b): b is Anthropic.ToolUseBlock => b.type === "tool_use"
  );

  if (toolUseBlocks.length === 0) {
    throw new Error(
      `Agent stuck at step ${step}: stop_reason=${response.stop_reason}, no tool calls`
    );
  }

  // Add assistant message
  messages.push({ role: "assistant", content: response.content });

  // Execute tools in parallel (when possible)
  const toolResults: ToolResult[] = await Promise.all(
    toolUseBlocks.map(async (toolUse) => {
      onStep?.(step, `executing_${toolUse.name}`);
      const result = await executeTool(toolUse.name, toolUse.input);
      return {
        type: "tool_result" as const,
        tool_use_id: toolUse.id,
        content: JSON.stringify(result),
      };
    })
  );

  // Add results and continue loop
  messages.push({ role: "user", content: toolResults });
}
```

```
  throw new Error(`Agent exceeded maxSteps=${maxSteps}`);
}

async function executeTool(
  name: string,
  input: Record<string, unknown>
): Promise<unknown> {
  // Your tool implementations here
  const handlers: Record<string, (input: Record<string, unknown>) => Promise<unknown>> = {
    read_file: async ({ path }) => {
      const fs = await import("fs/promises");
      return { content: await fs.readFile(String(path), "utf-8") };
    },
    write_file: async ({ path, content }) => {
      const fs = await import("fs/promises");
      await fs.writeFile(String(path), String(content), "utf-8");
      return { status: "written", path };
    },
    run_command: async ({ command }) => {
      const { exec } = await import("child_process");
      const { promisify } = await import("util");
      const execAsync = promisify(exec);
      const { stdout, stderr } = await execAsync(String(command), {
        timeout: 30000,
      });
      return { stdout, stderr };
    },
  };

  const handler = handlers[name];
  if (!handler) {
    return { error: `Unknown tool: ${name}` };
  }

  try {
    return await handler(input);
  } catch (error) {
    return {
      error: error instanceof Error ? error.message : String(error),
      tool: name,
    };
  }
}
```

Termination: The Most Important Thing You Are Not Thinking About

Every production agent system needs explicit termination logic. "The model decides when to stop" is not sufficient.

The model can get into loops. The model can decide it needs to do far more than the task required. The model can hallucinate tool results and keep going based on false beliefs. Any of these can run until you hit a context limit, a token budget, or a credit limit.

Minimum termination controls for any production agent:

```
class AgentTerminationManager:
    def __init__(
        self,
        max_steps: int = 20,
        max_duration_seconds: int = 300,
        max_input_tokens: int = 500_000,
        max_cost_usd: float = 1.00,
    ):
        self.max_steps = max_steps
        self.max_duration = max_duration_seconds
        self.max_tokens = max_input_tokens
        self.max_cost = max_cost_usd

        self.step_count = 0
        self.start_time = time.time()
        self.tokens_used = 0
        self.cost_accumulated = 0.0

    def check(self, tokens_this_step: int, cost_this_step: float) -> None:
        """Call before each step. Raises if any limit is exceeded."""
        self.step_count += 1
        self.tokens_used += tokens_this_step
        self.cost_accumulated += cost_this_step

        if self.step_count > self.max_steps:
            raise AgentLimitExceeded(f"Exceeded max steps ({self.max_steps})")

        elapsed = time.time() - self.start_time
        if elapsed > self.max_duration:
            raise AgentLimitExceeded(
                f"Exceeded max duration ({self.max_duration}s, ran {elapsed:.0f}s)"
            )

        if self.tokens_used > self.max_tokens:
            raise AgentLimitExceeded(
                f"Exceeded token budget ({self.tokens_used:,} > {self.max_tokens:,})"
            )
```

```
        if self.cost_accumulated > self.max_cost:
            raise AgentLimitExceeded(
                f"Exceeded cost budget (${self.cost_accumulated:.4f} > ${self.max_cost:.2f})"
            )
```

Set the limits tighter than you think you need. You can always raise them. You cannot un-spend tokens.

Key Takeaways

- An AI agent is a language model in a loop with tools — the loop runs until a goal is met or a limit is reached
- The agent loop: **Observe → Plan → Execute → Evaluate → repeat**
- Language models are stateless between calls; "memory" is an engineering choice, not a model feature
- Three architectures: single agent (simple, focused), multi-agent orchestrator+specialists (complex tasks), mesh/swarm (parallel high-throughput)
- Tool design is critical: atomic, well-typed, with safeguards on destructive operations, and observable via logging
- The mesh monitoring war story illustrates the most common agent overuse: using an expensive AI for tasks better served by cheap, purpose-built tools
- **Always implement termination limits**: max steps, max duration, max tokens, max cost. "The model decides when to stop" is not production-safe

---

Next: Chapter 3 — Choosing Your Model: A DevOps Engineer's Guide

---

Chapter 3: Choosing Your Model: A DevOps Engineer's Guide

> "Choosing the wrong model is not a technical mistake. It is a budget mistake." — Lesson learned after routing infrastructure debugging tasks to Gemini Flash

---

The Model Selection Problem

In 2024, choosing a model was relatively simple. There were two serious options: GPT-4 or Claude 3. You picked one based on vibes, API reliability, and whether your company had an existing vendor relationship.

In March 2026, you have approximately forty production-grade models across six providers, with meaningfully different capability profiles, price points, context window sizes, and failure modes. The wrong choice costs real money. The right choice can reduce your inference costs by 90% with no quality degradation.

This chapter gives you the decision framework I use for my own infrastructure. It is opinionated, based on real workloads, and honest about where each model family excels and where it disappoints.

The Pricing Landscape (March 2026)

Before decisions, data. All prices are per million tokens (MTok), input/output respectively.

Claude 4.x — Anthropic

Model	Input	Output	Context	Notes
Opus 4.6	$5	$25	200K (1M beta)	Best reasoning, mandatory for complex agents
Sonnet 4.6	$3	$15	200K	Best speed/intelligence balance

Model	Input	Output	Context	Notes
Haiku 4.5	$1	$5	200K	Fastest, highest throughput

Anthropic's pricing structure rewards the right choice for the right task. Opus is 5x the input cost of Haiku. If you are using Opus for tasks that Haiku can handle, you are burning 5x your budget for no quality gain.

Important note on context pricing: Unlike some providers, Anthropic does not currently charge a premium for long-context requests within the standard window. Prompt caching (beta) can reduce repeated context costs by up to 90% — critical for agent workloads where the system prompt hits every request.

GPT Series — OpenAI

Model	Input	Output	Context	Notes
GPT-4o	$2.50	$10	128K	Strong all-rounder, wide ecosystem
GPT-4.1	$2	$8	1M context	No long-context premium — best value at scale
o3	$10	$40	200K	Reasoning model, expensive
o4-mini	$1.10	$4.40	200K	80% cheaper than o3, good reasoning
GPT-5.3-Codex	See OpenAI	See OpenAI	200K	Code-specialized, strong on implementation

OpenAI's GPT-4.1 is particularly interesting for agent workloads that need large context without paying the Gemini 2x multiplier for long contexts. The 1M context window at $2/MTok input is genuinely competitive.

o3 and o4-mini are reasoning models — they spend more compute "thinking" before responding. Use them for problems that benefit from extended deliberation: complex planning, mathematical reasoning, multi-step logic. Do not use them for simple tasks; you pay for reasoning whether or not the task needs it.

Gemini — Google

Model	Input	Output	Context	Notes
2.5 Pro	$1.25	$10	1M	Long-context premium: 2x over 200K tokens
2.5 Flash	$0.30	$2.50	1M	Best cost/performance ratio available
2.5 Flash-Lite	~$0.10	~$0.40	1M	Ultra-cheap, simpler tasks

The Gemini 2x multiplier over 200K tokens is a critical detail for agent workloads. If your agent accumulates context over multiple turns — conversation history, tool results, memory — you can blow past 200K quickly. A request at 300K tokens costs 2x what you might estimate from the base rate.

Gemini Flash is the price-performance champion for high-volume workloads where intelligence requirements are moderate. Content classification, formatting, extraction, simple generation: Flash handles all of it at 1/10th the cost of Opus.

The $500 incident I mentioned in Chapter 1 used Gemini 1.5 Pro. The combination of verbose output, long contexts, the 2x multiplier, and a runaway loop created the bill. Flash would have been cheaper and — for that particular task (image metadata generation) — equally capable.

Open Source / Self-Hosted / Third-Party

Model	Input	Output	Context	Notes
DeepSeek R1	$0.55	$2.19	128K	MoE 671B, strong reasoning
DeepSeek V3.2	$0.14	$0.28	64K	Ultra-cheap, surprisingly capable
Llama 4 Maverick	$0.27	$0.85	1M	Meta, large context, competitive quality
Mistral Nemo	$0.02	$0.04	128K	Ultralight, simple tasks only

DeepSeek and Llama have matured significantly since 2024. For cost-sensitive workloads where you can tolerate slightly lower reliability and occasionally need to debug model-specific quirks, these represent genuine savings.

A practical note: these models are available through providers like Together AI, Groq, and Fireworks AI. The API interfaces are typically OpenAI-compatible, which means your existing code may need minimal changes to swap providers. But the behavior is not identical — test thoroughly before switching production workloads.

How to Think About Model Selection

The mistake most people make is treating model selection as a question of quality: "which model is smartest?"

The right question is: "what is the minimum capability required to complete this task reliably, and what is the cheapest model that meets that bar?"

The Capability Ladder

Think of models as a ladder where each rung costs more but provides more capability:

```
CAPABILITY vs COST LADDER (March 2026)

$25/MTok out    Claude Opus 4.6    Complex reasoning, ambiguous tasks,
                                       multi-step planning, infrastructure
                                       decisions, security analysis

$15/MTok out    Claude Sonnet 4.6   Balanced tasks, code review, content
                 GPT-4.1             generation, multi-tool agents where
                                      reliability matters

$10/MTok out    GPT-4o               Strong general tasks, wide tool
                 Gemini 2.5 Pro       ecosystem, familiar to many teams

 $5/MTok out    Claude Haiku 4.5    High-throughput, routing, classification,
                 o4-mini              simple agents, fast response needed

 $2.50/MTok     Gemini 2.5 Flash    Volume tasks, content processing,
                 DeepSeek R1          extraction, formatting

 $0.28/MTok     DeepSeek V3.2       Ultra-cheap, simple tasks, drafts,
                 Llama 4 Maverick     non-critical classification

 $0.04/MTok     Mistral Nemo        Trivial tasks only. Evaluate carefully.
```

The art is placing each task on the right rung.

The Selection Matrix

I use a four-question decision tree for every new agent or task:

Question 1: Does this require genuine reasoning or judgment?

"Reasoning" means the task requires weighing ambiguous information, making decisions with incomplete data, understanding context and nuance, or solving problems with no clear algorithmic solution.

Examples that require reasoning: - Diagnosing why an infrastructure component is behaving unexpectedly - Deciding whether a security vulnerability requires immediate action or can wait - Planning the implementation of a complex feature with multiple trade-offs - Writing code that requires understanding business logic, not just syntax

Examples that do NOT require reasoning: - Classifying a support ticket into categories - Extracting structured data from unformatted text - Translating content between formats (markdown to HTML, JSON to CSV) - Generating variations of existing content

If yes → start at Sonnet or Opus level. If no → start at Haiku or Flash level.

Question 2: What is the blast radius of a mistake?

If the agent is taking actions with real-world consequences (modifying files, calling APIs, sending messages, deploying services), a mistake costs more than just the token spend. It costs remediation time, potential data loss, potential user impact.

High blast radius tasks warrant higher model capability. Not because smarter models make zero mistakes — they do not — but because they make fewer mistakes on the reasoning and judgment calls that matter.

If blast radius is high → do not compromise on model capability. Opus for production infrastructure. If blast radius is low → aggressive cost optimization is safe.

Question 3: What is the request volume?

Volume multiplies cost. A model that costs 3x more per request costs 3x more when you run 10,000 requests per day. The economics shift dramatically at scale.

Rule of thumb: - Under 1,000 requests/day: model quality matters more than price - 1,000-100,000 requests/day: balance quality and price carefully - Over 100,000 requests/day: every $0.01/request is $1,000/day

At high volume, the correct approach is almost always to use the cheapest model that meets the quality bar, measure quality empirically, and upgrade if quality metrics fall below threshold.

Question 4: Does this need a long context window?

If your task requires feeding large amounts of data to the model — a full codebase, a large document, an extended conversation history — you need a model that supports it without penalty.

Watch out for: Gemini's 2x multiplier over 200K, providers that charge premiums for long contexts, and models with hard context limits that force you to truncate.

Best long-context options without penalty: GPT-4.1 (1M, no premium), Llama 4 Maverick (1M, cheap), Claude 4.x with prompt caching.

War Story: Routing Infrastructure Tasks to the Wrong Model

This is embarrassing to write, but it illustrates the failure mode clearly.

The Setup

In early 2025, when I was first standing up my multi-agent infrastructure, I built the CLIProxyAPI routing to default to Gemini 2.0 Flash for most tasks. Flash was cheap, fast, and capable enough for the bulk of what I needed. I had a rule that would route complex tasks to a more capable model, but I had not fully thought through the classification.

"Infrastructure management tasks" were classified as "complex" and routed to… Gemini 2.0 Pro. Not Claude Opus. Not even GPT-4.

The reason was cost. At the time, Gemini Pro was meaningfully cheaper than Opus. I convinced myself it was good enough for infrastructure debugging.

What Happened

Over three weeks, I accumulated a pattern: agents handling infrastructure issues kept getting stuck. Not catastrophically — they would not break things — but they would reach a decision point, produce a vague response, and hand the problem back to me with a summary that was correct but unhelpful.

A specific example: SG-02 had intermittent connection timeouts to the Redis instance. The agent was given access to server logs, Redis configuration, and network diagnostic tools. It ran through a standard diagnostic playbook, identified three possible causes, and concluded: "The issue may be related to network configuration, Redis timeout settings, or application connection pooling. Further investigation is recommended."

That is a correct answer. It is also useless. A good infrastructure engineer — or a capable agent — would have ranked those causes by likelihood given the log evidence, tested the most likely one, and either resolved it or explained precisely why not.

I switched the infrastructure routing to Claude Opus 4.6. Same logs, same tools, same prompt. The agent: 1. Identified that the timeout pattern was correlated with high CPU events on the application server (not Redis) 2. Recognized this as classic connection pool exhaustion under CPU pressure 3. Recommended specific pool size changes with reasoning 4. Drafted the configuration change for my review

Same task. Different outcome. The difference was not intelligence in the abstract — it was the model's ability to synthesize ambiguous evidence into a specific, actionable conclusion.

The Lesson

Gemini Flash (and similar cost-optimized models) are excellent at tasks with clear correct answers. Extraction, formatting, classification, generation from templates — these have objective correctness that cheaper models handle well.

Infrastructure debugging is not a task with a clear correct answer. It requires forming hypotheses, weighing evidence, considering what information is missing, and making judgment calls. This is where the capability gap between Gemini Flash and Claude Opus is most visible.

The cost difference is real: Gemini Flash at $0.30/MTok input vs. Opus at $5/MTok is a 16x difference. But if Gemini Flash costs me two hours of my own debugging time that Opus would have solved in fifteen minutes, the economics strongly favor Opus.

Always frame model cost against the total cost of the task, including engineer time.

War Story: The CLIProxyAPI Round-Robin with 15 Antigravity Accounts

Let me describe the actual routing system I run, because it illustrates several model selection principles in practice.

The Architecture

CLIProxyAPI is a local service running on my workstation that all agents use as their AI endpoint. It presents a unified OpenAI-compatible API interface and handles routing behind the scenes.

```
Agent Request


CLIProxyAPI (localhost:3000)

        Model routing rules



       Direct API                    Antigravity Pool
       (Claude, OpenAI)              (15 accounts, round-robin)


       Provider API              Gemini API (via Antigravity)

        Returns unified response format
```

Routing rules are configured per-model-prefix and per-task-tag:

```yaml
# CLIProxyAPI routing config (simplified)
routes:
  # Critical infrastructure tasks - direct to Claude API
  - match:
      model_prefix: "claude-opus"
      tags: ["infrastructure", "security", "deployment"]
    backend: direct_anthropic
    failover: null  # No failover - these tasks need Opus specifically

  # General agent tasks - Antigravity pool for Gemini
  - match:
      model_prefix: "gemini"
    backend: antigravity_pool
    failover: direct_google  # Fall back to direct Google if pool exhausted
```

```yaml
  # Monitoring and health checks - cheapest available
  - match:
      tags: ["health_check", "monitoring"]
    backend: antigravity_pool
    model_override: "gemini-2.5-flash-lite"  # Override whatever model was requested

  # Batch content jobs - pool with rate limiting
  - match:
      tags: ["batch", "content"]
    backend: antigravity_pool
    rate_limit:
      requests_per_minute: 20
      max_concurrent: 3

pool_config:
  antigravity:
    accounts: 15
    strategy: round_robin
    on_quota_exceeded:
      action: next_account
      max_retries: 3
      backoff_ms: 1000
    on_all_quota_exceeded:
      action: fail_with_error
      error_message: "All Antigravity accounts exhausted"
      notify: telegram
```

The pool management code:

```python
class AntigravityPool:
    def __init__(self, accounts: list[dict]):
        self.accounts = accounts
        self.current_index = 0
        self.quota_exceeded: set[str] = set()
        self.lock = threading.Lock()

    def get_next_account(self) -> dict | None:
        """Get next available account using round-robin."""
        with self.lock:
            attempts = 0
            while attempts < len(self.accounts):
                account = self.accounts[self.current_index]
                self.current_index = (self.current_index + 1) % len(self.accounts)

                if account["id"] not in self.quota_exceeded:
                    return account
                attempts += 1
```

```
            return None  # All accounts exhausted

    def mark_quota_exceeded(self, account_id: str) -> None:
        """Mark an account as quota-exceeded. Clears at midnight."""
        with self.lock:
            self.quota_exceeded.add(account_id)
            if len(self.quota_exceeded) == len(self.accounts):
                self._notify_all_exhausted()

    def _notify_all_exhausted(self) -> None:
        """Send Telegram alert when all accounts are exhausted."""
        send_telegram(
            "WARNING: All 15 Antigravity accounts quota exhausted. "
            "Gemini-routed requests will fail until quota resets."
        )
```

The Day All 15 Accounts Hit 100% Simultaneously

I mentioned this in Chapter 1 but want to go deeper here on the model selection aspect.

The batch job was running image metadata generation for 289 blog posts. The task was: given a blog post title and summary, generate three image search queries, a Gemini image generation prompt, alt text, and SEO metadata. Structured JSON output.

This is exactly the kind of task Flash is good at. Structured output, clear format, no ambiguous reasoning required. My model choice (Flash) was correct.

What was wrong was my consumption estimate. I had estimated ~400 tokens per request (input + output). The actual was ~1,200 tokens per request, due to:

- System prompt longer than I remembered (I had expanded it two weeks earlier)
- JSON output schema added to every request (~180 tokens)
- Gemini's verbose JSON generation (more whitespace, longer strings than estimated)
- The 2x multiplier kicking in because conversation history pushed me over 200K tokens midway through the batch

Actual consumption: 289 posts × 1,200 tokens × 4 API calls per post = **1,387,200 tokens**.

Expected consumption (my estimate): 289 × 400 × 4 = **462,400 tokens**.

3x the expected consumption across 15 accounts simultaneously. I had not accounted for the daily quota reset timing — all 15 accounts had reset the previous midnight, meaning all had identical quota headroom and all hit the ceiling at similar times.

The solution I implemented afterward:

```
def estimate_batch_cost(
    system_prompt: str,
    task_examples: list[str],
    output_examples: list[str],
    batch_size: int,
```

```
    model: str,
    context_multiplier: float = 1.0,
) -> dict:
    """
    Estimate cost before launching a batch job.
    Requires explicit confirmation if over threshold.
    """
    encoding = tiktoken.encoding_for_model("gpt-4")  # Close enough for estimation

    # Measure actual token counts from examples
    sample_input = system_prompt + "\n\n" + task_examples[0]
    sample_output = output_examples[0]

    input_tokens = len(encoding.encode(sample_input))
    output_tokens = len(encoding.encode(sample_output))

    # Add safety margin - real usage is usually higher than estimates
    input_tokens = int(input_tokens * 1.3)
    output_tokens = int(output_tokens * 1.3)

    # Apply context multiplier for long-context penalties (e.g., Gemini 2x)
    effective_input_tokens = input_tokens * context_multiplier

    # Get pricing
    pricing = MODEL_PRICING[model]
    cost_per_request = (
        (effective_input_tokens / 1_000_000) * pricing["input"] +
        (output_tokens / 1_000_000) * pricing["output"]
    )

    total_cost = cost_per_request * batch_size
    total_tokens = (input_tokens + output_tokens) * batch_size

    estimate = {
        "model": model,
        "batch_size": batch_size,
        "tokens_per_request": input_tokens + output_tokens,
        "cost_per_request_usd": round(cost_per_request, 6),
        "total_cost_usd": round(total_cost, 4),
        "total_tokens": total_tokens,
        "context_multiplier_applied": context_multiplier,
        "safety_margin_applied": "30%",
    }

    if total_cost > 5.00:
        print(f"\nWARNING: Estimated batch cost is ${total_cost:.2f}")
        print(f"  Model: {model}")
```

```
        print(f"  Requests: {batch_size:,}")
        print(f"  Tokens/request: {input_tokens + output_tokens:,}")
        confirm = input("\nProceed? [y/N]: ")
        if confirm.lower() != "y":
            raise UserCancelled("Batch job cancelled by user")

    return estimate
```

War Story: The Model Mismatch Bug (HTTP 400 INVALID_ARGUMENT)

This one took two hours to debug and was entirely self-inflicted.

The Setup

OpenClaw was configured to use `claude-opus-4-5-thinking` — a reasoning model variant I had been testing. The config file had been set manually and I had forgotten about it when I moved OpenClaw's API traffic through the CLIProxyAPI router.

The CLIProxyAPI router had a routing rule that said: if the model name starts with `claude`, route to the Anthropic API directly. Sensible.

But I had also been experimenting with using the Antigravity pool for some Claude traffic via a compatibility layer. The compatibility layer was configured to translate Claude API calls to Gemini API calls, allowing Gemini to serve Claude-formatted requests (at a significant discount).

What Happened

The routing logic ran as follows:

1. OpenClaw requests `claude-opus-4-5-thinking`
2. Router: starts with `claude` → direct Anthropic… except wait, the Antigravity compatibility flag was on
3. Router (with flag): translate Claude request to Gemini format → route to Antigravity pool
4. Translation layer: translates model name to `gemini-2.5-pro` (closest equivalent)
5. But: the original request included `reasoning_effort: "high"` parameter (Anthropic-specific for thinking models)
6. Gemini API receives request with `reasoning_effort` parameter
7. Gemini API returns HTTP 400 INVALID_ARGUMENT

The error message:

```
HTTP 400 Bad Request
{
  "error": {
    "code": 400,
    "message": "Request contains an unknown parameter: reasoning_effort",
    "status": "INVALID_ARGUMENT"
  }
```

}

The debugging path: 1. OpenClaw reports repeated 400 errors, cannot complete tasks 2. I check OpenClaw logs — 400 INVALID_ARGUMENT, model: gemini-2.5-pro 3. I check OpenClaw config — model set to `claude-opus-4-5-thinking` 4. I check CLIProxyAPI logs — translation is happening, `reasoning_effort` not being stripped 5. I check the translation code — it translates model names but does not strip provider-specific parameters

The fix:

```python
def translate_anthropic_to_gemini(request: dict) -> dict:
    """
    Translate Anthropic API request format to Gemini format.
    Must strip Anthropic-specific parameters that Gemini does not understand.
    """
    ANTHROPIC_ONLY_PARAMS = {
        "reasoning_effort",     # Thinking model control
        "betas",                # Beta feature flags
        "anthropic_version",    # API version header
        "thinking",             # Thinking block config
    }

    translated = {
        "model": ANTHROPIC_TO_GEMINI_MODELS.get(
            request.get("model", ""),
            "gemini-2.5-pro"  # Default
        ),
        "contents": translate_messages(request.get("messages", [])),
        "generationConfig": {
            "maxOutputTokens": request.get("max_tokens", 4096),
            "temperature": request.get("temperature", 1.0),
        },
    }

    # Warn if stripping reasoning-related params - behavior will differ
    stripped = []
    for param in ANTHROPIC_ONLY_PARAMS:
        if param in request:
            stripped.append(param)

    if stripped:
        logger.warning(
            f"Translation stripped Anthropic-specific params: {stripped}. "
            f"Behavior may differ from intended model."
        )

    return translated
```

The Larger Lesson

Provider-specific model parameters do not translate across providers. This sounds obvious in retrospect. It was not obvious when I was building the compatibility layer at 11 PM.

If you build any abstraction layer that routes between model providers, you must: 1. Strip provider-specific parameters before routing to a different provider 2. Log when you strip parameters (behavior changes silently otherwise) 3. Understand which parameters are provider-specific vs. universal 4. Test the translation layer with requests that include edge-case parameters

Provider-specific parameters to watch for:

Provider	Specific Parameters	Notes
Anthropic	`reasoning_effort`, `thinking`, `betas`, `anthropic_version`	Thinking model controls
OpenAI	`response_format` with strict mode, `store`, `prediction`	Not all Gemini-compatible
Google	`generation_config.response_mime_type`, `safety_settings`	Google-specific safety
OpenAI reasoning	`reasoning_effort` (o3/o4), `reasoning_summary`	Different meaning than Anthropic

---

Building a Model Selection Policy

Here is the actual policy I run, expressed as code:

```
from enum import Enum
from dataclasses import dataclass

class TaskType(Enum):
    # Requires complex reasoning, judgment, ambiguous inputs
    INFRASTRUCTURE_DEBUG = "infrastructure_debug"
    SECURITY_ANALYSIS = "security_analysis"
    ARCHITECTURE_PLANNING = "architecture_planning"
    COMPLEX_CODE_REVIEW = "complex_code_review"

    # Requires good capability, medium complexity
    CODE_GENERATION = "code_generation"
    CONTENT_WRITING = "content_writing"
    DATA_ANALYSIS = "data_analysis"
    AGENT_ORCHESTRATION = "agent_orchestration"

    # Routing, classification, simple decisions
    TASK_ROUTING = "task_routing"
    SENTIMENT_CLASSIFICATION = "sentiment_classification"
    FORMAT_CONVERSION = "format_conversion"
    HEALTH_CHECK_ANALYSIS = "health_check_analysis"
```

```
    # High-volume, simple, clear-correct-answer tasks
    METADATA_EXTRACTION = "metadata_extraction"
    TEMPLATE_GENERATION = "template_generation"
    TRANSLATION = "translation"
    SUMMARIZATION_SIMPLE = "summarization_simple"


@dataclass
class ModelPolicy:
    primary: str
    fallback: str | None
    max_tokens: int
    temperature: float
    notes: str


MODEL_POLICIES: dict[TaskType, ModelPolicy] = {
    # Critical reasoning tasks - no compromise
    TaskType.INFRASTRUCTURE_DEBUG: ModelPolicy(
        primary="claude-opus-4-6",
        fallback=None,  # No fallback - Opus or nothing
        max_tokens=8192,
        temperature=0.1,  # Low temp for deterministic diagnosis
        notes="Infrastructure decisions need best reasoning. Cost is secondary.",
    ),
    TaskType.SECURITY_ANALYSIS: ModelPolicy(
        primary="claude-opus-4-6",
        fallback=None,
        max_tokens=8192,
        temperature=0.1,
        notes="Security analysis errors have high blast radius.",
    ),
    TaskType.ARCHITECTURE_PLANNING: ModelPolicy(
        primary="claude-opus-4-6",
        fallback="claude-sonnet-4-6",
        max_tokens=16384,
        temperature=0.3,
        notes="Complex planning benefits from Opus. Sonnet fallback acceptable.",
    ),

    # Capable model, cost-aware
    TaskType.CODE_GENERATION: ModelPolicy(
        primary="claude-sonnet-4-6",
        fallback="gpt-4.1",
        max_tokens=8192,
        temperature=0.2,
        notes="Sonnet handles most code well. Opus for especially complex logic.",
```

```
    ),
    TaskType.CONTENT_WRITING: ModelPolicy(
        primary="claude-sonnet-4-6",
        fallback="gemini-2.5-pro",
        max_tokens=8192,
        temperature=0.7,
        notes="Higher temp for creative variation. Sonnet has good voice.",
    ),
    TaskType.AGENT_ORCHESTRATION: ModelPolicy(
        primary="claude-sonnet-4-6",
        fallback="gpt-4.1",
        max_tokens=4096,
        temperature=0.1,
        notes="Orchestration needs reliability. Low temp for consistent routing.",
    ),

    # Cheap and fast
    TaskType.TASK_ROUTING: ModelPolicy(
        primary="claude-haiku-4-5",
        fallback="gemini-2.5-flash",
        max_tokens=512,
        temperature=0.0,
        notes="Routing is classification. Cheapest capable model.",
    ),
    TaskType.HEALTH_CHECK_ANALYSIS: ModelPolicy(
        primary="gemini-2.5-flash",
        fallback="claude-haiku-4-5",
        max_tokens=1024,
        temperature=0.0,
        notes="Health checks need speed and low cost, not intelligence.",
    ),

    # Batch / high-volume
    TaskType.METADATA_EXTRACTION: ModelPolicy(
        primary="gemini-2.5-flash",
        fallback="deepseek-v3-2",
        max_tokens=1024,
        temperature=0.1,
        notes="High volume, clear correct answers. Flash is ideal.",
    ),
    TaskType.TEMPLATE_GENERATION: ModelPolicy(
        primary="gemini-2.5-flash",
        fallback="deepseek-v3-2",
        max_tokens=2048,
        temperature=0.3,
        notes="Template fill is mechanical. Minimize cost.",
    ),
```

```
}


def get_model_for_task(
    task_type: TaskType,
    budget_remaining_usd: float | None = None,
    prefer_fast: bool = False,
) -> str:
    """
    Return the appropriate model for a given task type.
    Adjusts based on remaining budget if provided.
    """
    policy = MODEL_POLICIES[task_type]

    # Budget override - if budget is tight, fall back if possible
    if budget_remaining_usd is not None and budget_remaining_usd < 0.50:
        if policy.fallback:
            logger.warning(
                f"Budget low (${budget_remaining_usd:.2f}), "
                f"using fallback {policy.fallback} for {task_type.value}"
            )
            return policy.fallback

    # Speed override - use a faster model if real-time response needed
    if prefer_fast and task_type in FAST_OVERRIDE_ALLOWED:
        return FAST_MODEL_ALTERNATIVES.get(policy.primary, policy.primary)

    return policy.primary


# Tasks where speed override is acceptable (quality difference tolerable)
FAST_OVERRIDE_ALLOWED = {
    TaskType.CONTENT_WRITING,
    TaskType.FORMAT_CONVERSION,
    TaskType.TRANSLATION,
    TaskType.SUMMARIZATION_SIMPLE,
}
```

The Open Source Reality Check

DeepSeek, Llama, and Mistral deserve honest assessment. The hype around open source models sometimes oversells their production readiness.

What open source models do well: - Cost-sensitive, high-volume tasks where occasional errors are acceptable - Tasks with clear correct answers (extraction, classification, formatting) - Private deployment where data cannot leave your infrastructure - Experimentation and development where

production reliability is not required

Where they fall short for agent workloads: - Tool use reliability is lower than frontier models at equivalent quality levels - Instruction following consistency is more variable - Long-context coherence degrades faster - Debugging model-specific quirks adds engineering overhead

DeepSeek R1 (MoE 671B at $0.55/$2.19) is genuinely competitive with mid-tier frontier models for reasoning tasks. I have used it for code generation tasks where I needed privacy (data not leaving my infrastructure) and the quality was acceptable.

DeepSeek V3.2 ($0.14/$0.28) is the most extreme cost option I use in production. I use it for tasks where "pretty good" is sufficient and volume is very high. It fails more often than Gemini Flash on tool use, so I never use it for agents that need reliable tool calling.

Mistral Nemo ($0.02/M) is so cheap it is almost free. I use it for internal classification tasks where I can tolerate errors and human review catches mistakes. I do not use it for anything customer-facing or with side effects.

Monitoring Your Model Selection Over Time

Model selection is not a one-time decision. Models improve, pricing changes, your workloads evolve. Build measurement into your system from day one.

```
class ModelPerformanceTracker:
    """
    Track per-model quality metrics to inform selection decisions.
    Store in a time-series format for trend analysis.
    """

    def __init__(self, db_path: str = "~/.agent/model_metrics.db"):
        self.db = sqlite3.connect(db_path)
        self._init_schema()

    def _init_schema(self):
        self.db.execute("""
            CREATE TABLE IF NOT EXISTS model_runs (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                timestamp TEXT NOT NULL,
                model TEXT NOT NULL,
                task_type TEXT NOT NULL,
                input_tokens INTEGER,
                output_tokens INTEGER,
                duration_ms INTEGER,
                success BOOLEAN,
                error_type TEXT,
                quality_score REAL,  -- Human-rated or automated eval
                cost_usd REAL,
                notes TEXT
```

```
            )
        """)
        self.db.commit()

    def record(
        self,
        model: str,
        task_type: str,
        input_tokens: int,
        output_tokens: int,
        duration_ms: int,
        success: bool,
        cost_usd: float,
        quality_score: float | None = None,
        error_type: str | None = None,
    ) -> None:
        self.db.execute("""
            INSERT INTO model_runs
            (timestamp, model, task_type, input_tokens, output_tokens,
             duration_ms, success, error_type, quality_score, cost_usd)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """, (
            datetime.utcnow().isoformat(),
            model, task_type, input_tokens, output_tokens,
            duration_ms, success, error_type, quality_score, cost_usd,
        ))
        self.db.commit()

    def get_model_summary(self, days: int = 30) -> list[dict]:
        """Return per-model performance summary for the last N days."""
        cutoff = (datetime.utcnow() - timedelta(days=days)).isoformat()
        cursor = self.db.execute("""
            SELECT
                model,
                task_type,
                COUNT(*) as runs,
                AVG(CASE WHEN success THEN 1.0 ELSE 0.0 END) as success_rate,
                AVG(quality_score) as avg_quality,
                AVG(duration_ms) as avg_latency_ms,
                SUM(cost_usd) as total_cost_usd,
                AVG(cost_usd) as avg_cost_usd
            FROM model_runs
            WHERE timestamp > ?
            GROUP BY model, task_type
            ORDER BY total_cost_usd DESC
        """, (cutoff,))
        return [dict(zip([col[0] for col in cursor.description], row))
```

```
                for row in cursor.fetchall()]
```

Run this report weekly. You will find surprises: models you thought were performing well have lower success rates than you realized. Models you under-estimated are handling tasks better than the more expensive alternatives. Price changes that make your current selection suboptimal.

Quick Reference: Selection Rules

For the moment when you need a fast answer:

```
DECISION TREE: Which Model?

Is it infrastructure/security/architecture?
  YES → Claude Opus 4.6 (no compromise)
  NO ↓

Does it require judgment on ambiguous data?
  YES → Claude Sonnet 4.6 or GPT-4.1
  NO ↓

Is it code generation or complex writing?
  YES → Claude Sonnet 4.6 (or GPT-5.3-Codex for pure code)
  NO ↓

Is it routing, classification, or simple decisions?
  YES → Claude Haiku 4.5 or Gemini 2.5 Flash
  NO ↓

Is it high-volume, clear-correct-answer, batch?
  YES → Gemini 2.5 Flash (or DeepSeek V3.2 for extreme cost sensitivity)
  NO ↓

Is privacy/on-premise required?
  YES → DeepSeek R1 or Llama 4 Maverick (self-hosted)
  NO → Revisit earlier questions - you likely fit one of those categories
```

When in doubt, run a small experiment. Take 20 representative examples from your actual workload. Run them through the cheap model and the expensive model. Compare outputs. The quality difference is often smaller — or larger — than you expect, and real data beats speculation.

Key Takeaways

- Model selection is a cost-quality optimization problem, not a "which is smartest" question
- The selection matrix: complexity of reasoning, blast radius of mistakes, request volume, context window needs

- **Claude Opus 4.6 is mandatory for infrastructure debugging, security analysis, and complex multi-step agents** — the capability gap is real and the time cost of mistakes exceeds the token cost difference
- **Gemini 2.5 Flash is the cost-performance champion** for high-volume, clear-correct-answer tasks — but watch the 2x multiplier over 200K tokens
- **GPT-4.1** has the best long-context economics (1M tokens, no premium) for workloads that need large windows
- The routing + Antigravity pool architecture works but requires accurate quota estimation and circuit breakers on the consumer side
- Model mismatch bugs (provider-specific parameters crossing API boundaries) are a real failure mode — build parameter stripping into any translation layer
- Measure model performance empirically with real workloads — surprises in both directions are common

---

Next: Chapter 4 — Designing Agent Systems That Survive Contact With Reality

---

PART 2: ARCHITECTURE

Chapter 4: Agent Architecture Patterns

> "The right architecture isn't the most clever one. It's the one that still works at 3am when something catches fire."

Introduction

When I first started deploying AI agents in production, I made the same mistake everyone makes: I treated them like microservices. Give them an API, point them at some tools, done. Ship it.

That worked fine until the first time an agent needed to coordinate with another agent. Then it needed to monitor infrastructure. Then it needed to recover from its own failures. Then I had three agents watching each other across three servers and none of them agreed on who was in charge.

This chapter is about the architectural patterns I've learned the hard way. Not the theoretical ones from papers. The ones that survived contact with production.

We'll cover five main patterns, ordered roughly from simple to complex:

1. **Single Agent** — one agent, one job
2. **Multi-Agent Orchestration** — a coordinator with sub-agents
3. **Agent Mesh** — peers that monitor and heal each other
4. **Event-Driven Agents** — agents woken by real-time events
5. **Hybrid Architectures** — mixing patterns in production

For each pattern, I'll give you the real implementation details, the failure modes I ran into, and what I'd do differently.

Pattern 1: Single Agent (Claude Code Standalone)

This is where everyone starts. One agent, one session, one task.

```
User → Claude Code → Tools (bash, read, write, git) → Output
```

When It Works

Single-agent is underrated. For a surprising number of tasks, it's exactly right:

- Refactoring a codebase
- Writing and running a batch migration script
- Debugging a production incident (with read-only tools)
- Generating boilerplate or documentation

The simplicity is a feature. One process, one log file, one thing to debug when something goes wrong.

The Real Constraint: Context Window

The context window is your single agent's entire working memory. Claude Opus 4.6 has 200K tokens. Sounds like a lot until you realize:

```
System prompt:         ~2,000 tokens
Tools definitions:     ~1,000 tokens
Conversation history: grows over time
Code being edited:     varies wildly
Output buffer:         ~8,000 tokens
```

A session that starts at 10K tokens used can hit 150K after an hour of serious work. When it approaches the limit, the agent's behavior degrades. It starts forgetting context from earlier in the conversation. It makes mistakes it wouldn't have made at the start.

I've had Claude Code sessions reach 97% quota utilization and get killed with signal 9. No graceful shutdown, no checkpoint. The work just stops.

Rule of thumb: Plan for a single-agent session to handle no more than what fits comfortably in 100K tokens of conversation. Beyond that, you need a different architecture.

Single Agent Anti-Patterns

The super-prompt failure. Early on I tried to write one massive prompt that would handle an entire complex deployment: update the database schema, migrate the data, update the application code, restart services, run smoke tests. The agent would start well, then hallucinate midway through because it had too much to track.

The fix was to split into four sequential prompts with explicit handoffs: 1. `Phase 1: Analyze current state and plan migration` 2. `Phase 2: Execute database changes` (after reviewing phase 1 output) 3. `Phase 3: Update application code` 4. `Phase 4: Deploy and validate`

Each phase starts fresh with targeted context. The agent knows exactly what it needs to know.

The relative path trap. Single agents run in sessions that die. If you reference `./configs/production.yaml`, that works fine in the current session. But if the agent session gets killed and restarted, or if you spawn a sub-agent, the working directory might be different.

Always use absolute paths in agent instructions. Always. I've burned hours debugging failures that were just `./config` not resolving from a different working directory.

```
# Bad - will break across sessions
./scripts/deploy.sh

# Good - works everywhere
/home/hieudc/project/scripts/deploy.sh
```

Pattern 2: Multi-Agent Orchestration

When single-agent hits its limits, the natural next step is an orchestrator-subagent model. One agent coordinates; multiple specialized agents do the work.

```
User
  ↓
Orchestrator Agent
     Sub-agent A (code changes)
     Sub-agent B (testing)
     Sub-agent C (documentation)
```

How It Works in Practice

The orchestrator receives a complex task and breaks it into parallel or sequential workstreams. Each sub-agent gets a focused context: just what it needs to complete its specific piece.

This is how I had five agents write five parts of a book simultaneously. The orchestrator defined the chapters, assigned one to each agent with the same style guide, ran them in parallel, and merged the outputs. Twenty-five minutes for a full draft. Doing it sequentially in one session would have taken hours and likely hit context limits.

Sub-Agent Context Budget

Each sub-agent gets its own fresh 200K token context. This is the key advantage over single-agent for large tasks: you're not fighting a shrinking context across a long session. Each agent starts clean.

The constraint is task design. You must structure each sub-agent's task so it can complete within: - Its own context window (200K) - Without needing real-time coordination with other agents (unless you explicitly design for it)

Practical rule: Design sub-agent tasks to be independent. If agent B needs output from agent A, don't run them in parallel—chain them sequentially, with A's output passed explicitly in B's prompt.

The Orchestration Pattern I Use

```
# Sub-Agent Prompt Template

## Your Role
You are [specific role] for [specific task].
```

```
## Context
[Relevant background, exactly what this agent needs]

## Task
[Specific, bounded deliverable]

## Output Format
[Exactly what to produce and where to put it]

## Constraints
- Work only in [specific directory/scope]
- Do NOT modify [list of things to leave alone]
- If blocked, write blockers to [specific file] and stop

## Absolute Paths
- Work dir: /home/hieudc/project/
- Output: /home/hieudc/project/output/agent-N/
```

The constraints section is critical. Sub-agents are capable and eager. Without explicit boundaries, they'll "help" by doing things you didn't ask for. I've had agents restructure directories, update config files, install dependencies—all while successfully completing the assigned task and completely breaking the environment for the next agent.

War Story: The 15-Agent Pileup

I once spawned 15 sub-agents in parallel to process 15 chapters simultaneously. Each agent was supposed to rewrite one chapter. What I didn't account for: three of them needed the same shared config file. They all wrote to it simultaneously. The resulting file was corrupted. All three chapters failed, and I spent forty minutes untangling the merge conflict.

Lesson: Identify shared resources before spawning parallel agents. Either give each agent a private copy, or serialize the agents that touch shared resources.

The Cleanup Problem

Sub-agent sessions are not free. They consume quota. If an agent session hangs or gets orphaned—maybe it got stuck on a network call, maybe it lost connectivity—it keeps its tmux session alive and keeps burning through your API quota.

I discovered this the hard way when I found seven orphaned tmux sessions quietly running. None of them were doing anything useful. They were just alive, and their heartbeat prompts kept firing.

```
# Check for orphaned agent sessions
tmux ls | grep -E "agent|sub"

# Kill them
tmux kill-session -t agent-3
```

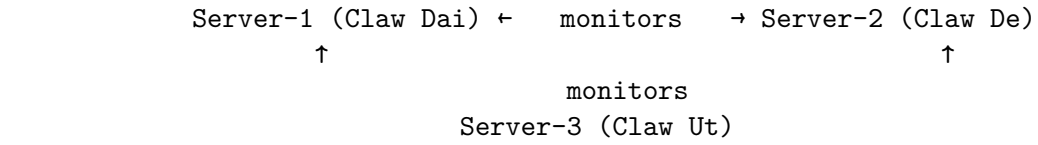
After this incident I added a cleanup policy: any sub-agent spawned programmatically gets a

`cleanup: "delete"` flag and a maximum wall-clock timeout. If it doesn't complete within the timeout, it gets killed.

Pattern 3: Agent Mesh

This is where it gets interesting—and where I spent three weeks iterating on a design that actually works.

An agent mesh is a group of peer agents that monitor each other and can recover each other's failures. No single agent is special. Any agent can observe any other agent's health. Any agent can attempt to fix any other agent.



In my production setup, I run three servers—Server-1 in Singapore, Server-2 in the same datacenter, Server-3 in Vietnam. Each runs an OpenClaw gateway agent. Each runs the same mesh monitor script on a 3-minute heartbeat. Each agent can restart any other agent over SSH.

Evolution: v1 → v2.3

Getting this right took three major revisions. Let me walk through what broke and why.

v1: No Coordination

First version was simple: if agent A sees agent B is down, agent A restarts B. If agent C also sees B is down at the same time, agent C also tries to restart B. Both succeed, or they conflict. Either way, you get duplicate restart attempts and noisy logs.

Fine for occasional failures. Terrible when a network hiccup makes all agents see each other as down simultaneously and they all start restarting everything.

v2.2: Leader Election by IP

Added leader election: the agent with the lowest IP address is the "leader" and is the only one allowed to execute recoveries.

This worked until I realized Server-2 (10.10.0.3) could never be the leader, because Server-1 (10.10.0.2) and Server-3 (10.10.0.1) both had lower IPs. Server-2 would detect failures and correctly diagnose them, but never trigger recovery. It was a passenger on its own mesh.

The question that exposed this bug: "What's Server-2's role in the mesh?" The answer—"it watches but never acts"—made the problem obvious.

v2.3: Pure Flock

The solution was to drop leader election entirely and use kernel-level file locking (`flock`) directly.

How flock-based coordination works:

1. Each agent, when it detects a failure, tries to acquire an exclusive lock on the target machine's recovery lock file.
2. The lock is acquired via SSH: the remote machine holds the file descriptor.
3. Because the lock is held by the SSH process, it's automatically released when the SSH session ends—whether that's normal completion, crash, or network drop.
4. The kernel guarantees that only one process holds the lock at a time, with no race conditions.

```
# The critical insight: run diagnosis AND fix inside a single SSH session
# Lock is held for the entire operation, released on session exit

ssh_with_lock() {
    local target_host="$1"
    local target_agent="$2"

    ssh -o ConnectTimeout=10 "$target_host" bash << 'REMOTE_SCRIPT'
        LOCK_FILE="/tmp/openclaw-recovery.lock"

        # Try to acquire lock - non-blocking
        exec 200>"$LOCK_FILE"
        if ! flock -n 200; then
            echo "LOCK_BUSY: another agent is already recovering"
            exit 0  # Not an error, just skip
        fi

        # Lock acquired - diagnose and fix within this session
        # Lock releases automatically when this SSH session ends
        diagnose_and_fix
REMOTE_SCRIPT
}
```

The beauty of this approach:

- **No orphan locks.** SSH dies → bash process dies → file descriptor closes → kernel releases lock. No cleanup needed.
- **No leader election complexity.** Any agent can fix any agent. First one to grab the lock wins.
- **Atomic at kernel level.** No gap between "check if locked" and "acquire lock."
- **Natural backpressure.** If a fix takes 5 minutes, no other agent wastes resources trying the same fix during those 5 minutes.

The Full Mesh Monitor Script

Here's the production script (v2.3, simplified for clarity):

```
#!/bin/bash
# agent-mesh-monitor.sh
# Version 2.3 - Pure flock coordination, no leader election
# Runs on each server, monitors the other two

set -euo pipefail
```

```
AGENT_NAME="${OPENCLAW_AGENT_NAME:-unknown}"
FAIL_THRESHOLD=2
HEARTBEAT_DIR="/home/hieudc/.openclaw/workspace"
TELEGRAM_TOKEN="${TELEGRAM_BOT_TOKEN}"
TELEGRAM_CHAT="${TELEGRAM_CHAT_ID}"
LOG_FILE="/tmp/mesh-monitor-${AGENT_NAME}.log"

# Peer configuration
declare -A PEERS=(
    ["claw-dai"]="10.10.0.2"
    ["claw-de"]="10.10.0.3"
    ["claw-ut"]="10.10.0.1"
)

declare -A FAIL_COUNTS=()
for peer in "${!PEERS[@]}"; do
    FAIL_COUNTS[$peer]=0
done

log() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $*" | tee -a "$LOG_FILE"
}

send_telegram() {
    local message="$1"
    curl -s -X POST \
        "https://api.telegram.org/bot${TELEGRAM_TOKEN}/sendMessage" \
        -d "chat_id=${TELEGRAM_CHAT}" \
        -d "text=${message}" \
        -d "parse_mode=Markdown" > /dev/null 2>&1 || true
}

check_heartbeat() {
    local peer_name="$1"
    local peer_ip="${PEERS[$peer_name]}"
    local heartbeat_file="${HEARTBEAT_DIR}/heartbeats/${peer_name}.md"

    # Check heartbeat file was updated recently
    if [[ ! -f "$heartbeat_file" ]]; then
        log "WARN: No heartbeat file for $peer_name"
        return 1
    fi

    local last_modified
    last_modified=$(stat -c %Y "$heartbeat_file" 2>/dev/null || echo 0)
    local now
    now=$(date +%s)
```

```
    local age=$(( now - last_modified ))

    # Heartbeat should be updated every 30 minutes (1800 seconds)
    # Give 2x buffer for network/timing variance
    if [[ $age -gt 3600 ]]; then
        log "STALE: $peer_name heartbeat is ${age}s old"
        return 1
    fi

    # Also try a simple TCP connectivity check
    if ! timeout 5 bash -c "cat < /dev/null > /dev/tcp/${peer_ip}/22" 2>/dev/null; then
        log "UNREACHABLE: Cannot reach $peer_name at $peer_ip:22"
        return 1
    fi

    return 0
}

diagnose_peer() {
    local peer_name="$1"
    local peer_ip="${PEERS[$peer_name]}"

    log "DIAGNOSE: Checking $peer_name ($peer_ip)"

    # Run diagnosis on remote machine
    local diagnosis
    diagnosis=$(ssh -o ConnectTimeout=10 -o StrictHostKeyChecking=no \
        "hieudc@${peer_ip}" bash << 'EOF' 2>/dev/null || echo "SSH_FAILED")

        issues=()

        # Check OpenClaw gateway
        if ! systemctl --user is-active openclaw-gateway > /dev/null 2>&1; then
            issues+=("gateway:inactive")
        fi

        # Check CLIProxyAPI
        if ! systemctl is-active cliproxyapi > /dev/null 2>&1; then
            issues+=("cliproxyapi:inactive")
        fi

        # Check disk space
        disk_used=$(df / --output=pcent | tail -1 | tr -d '% ')
        if [[ $disk_used -gt 90 ]]; then
            issues+=("disk:${disk_used}%")
        fi
```

```
        # Check memory
        mem_available=$(free -m | awk '/^Mem:/{print $7}')
        if [[ $mem_available -lt 512 ]]; then
            issues+=("memory:${mem_available}MB")
        fi

        if [[ ${#issues[@]} -eq 0 ]]; then
            echo "HEALTHY"
        else
            echo "ISSUES:$(IFS=','; echo "${issues[*]}")"
        fi
EOF

    echo "$diagnosis"
}

attempt_recovery() {
    local peer_name="$1"
    local peer_ip="${PEERS[$peer_name]}"
    local issues="$2"

    log "RECOVERY: Attempting to fix $peer_name, issues: $issues"

    # Try to acquire the recovery lock on the target machine
    # If another agent already has the lock, we skip
    local result
    result=$(ssh -o ConnectTimeout=15 -o StrictHostKeyChecking=no \
        "hieudc@${peer_ip}" bash << REMOTE_EOF 2>&1 || echo "SSH_FAILED")

        LOCK_FILE="/tmp/openclaw-recovery.lock"
        exec 200>"\$LOCK_FILE"

        if ! flock -n 200; then
            echo "LOCK_BUSY"
            exit 0
        fi

        # We have the lock - attempt fixes
        fixed=()

        if [[ "$issues" == *"gateway:inactive"* ]]; then
            systemctl --user start openclaw-gateway
            sleep 5
            if systemctl --user is-active openclaw-gateway > /dev/null 2>&1; then
                fixed+=("gateway:restarted")
            fi
        fi
```

```
        if [[ "$issues" == *"cliproxyapi:inactive"* ]]; then
            systemctl start cliproxyapi
            sleep 5
            if systemctl is-active cliproxyapi > /dev/null 2>&1; then
                fixed+=("cliproxyapi:restarted")
            fi
        fi

        if [[ "\${#fixed[@]}" -gt 0 ]]; then
            echo "FIXED:\$(IFS=','; echo "\${fixed[*]}")"
        else
            echo "NO_FIX"
        fi

        # Lock releases here when SSH session ends
REMOTE_EOF

    echo "$result"
}

monitor_loop() {
    while true; do
        for peer_name in "${!PEERS[@]}"; do
            # Skip monitoring yourself
            if [[ "$peer_name" == "$AGENT_NAME" ]]; then
                continue
            fi

            if check_heartbeat "$peer_name"; then
                # Healthy - reset fail count
                FAIL_COUNTS[$peer_name]=0
                log "OK: $peer_name is healthy"
            else
                # Failed - increment counter
                FAIL_COUNTS[$peer_name]=$(( ${FAIL_COUNTS[$peer_name]} + 1 ))
                count=${FAIL_COUNTS[$peer_name]}

                log "FAIL $count/$FAIL_THRESHOLD: $peer_name not responding"

                if [[ $count -ge $FAIL_THRESHOLD ]]; then
                    log "CRITICAL: $peer_name failed $FAIL_THRESHOLD times - triggering recover
                    send_telegram "  *CRITICAL* [$AGENT_NAME]: $peer_name is DOWN (fail $count/

                    local diagnosis
                    diagnosis=$(diagnose_peer "$peer_name")
                    log "DIAGNOSIS for $peer_name: $diagnosis"
```

```
                    if [[ "$diagnosis" != "HEALTHY" && "$diagnosis" != "SSH_FAILED" ]]; then
                        local recovery_result
                        recovery_result=$(attempt_recovery "$peer_name" "$diagnosis")
                        log "RECOVERY RESULT for $peer_name: $recovery_result"

                        if [[ "$recovery_result" == *"FIXED"* ]]; then
                            send_telegram "  [$AGENT_NAME]: Fixed $peer_name - $recovery_result
                            FAIL_COUNTS[$peer_name]=0
                        elif [[ "$recovery_result" == "LOCK_BUSY" ]]; then
                            log "INFO: Another agent is already recovering $peer_name"
                            FAIL_COUNTS[$peer_name]=0  # Don't keep counting, assume other age
                        else
                            send_telegram "  [$AGENT_NAME]: Could not fix $peer_name - $recover
                        fi
                    fi
                fi
            fi
        done

        # Wait before next check cycle
        sleep 180  # 3 minutes
    done
}

log "Starting mesh monitor as $AGENT_NAME"
monitor_loop
```

Deployment: systemd Timer Instead of Cron

After the token optimization crisis (more on that in Chapter 5), I moved from running the mesh monitor through OpenClaw to running it as a pure systemd timer. This means zero AI tokens consumed per check cycle.

```
# /etc/systemd/system/mesh-monitor.service
[Unit]
Description=Agent Mesh Monitor
After=network.target

[Service]
Type=simple
User=hieudc
Environment="OPENCLAW_AGENT_NAME=claw-dai"
Environment="TELEGRAM_BOT_TOKEN=xxx"
Environment="TELEGRAM_CHAT_ID=821796713"
ExecStart=/home/hieudc/.openclaw/workspace/scripts/agent-mesh-monitor.sh
Restart=always
RestartSec=30
```

```
[Install]
WantedBy=multi-user.target
```

```
# No timer needed - the script loops internally
# But if you want periodic runs instead:
# /etc/systemd/system/mesh-monitor.timer
[Unit]
Description=Agent Mesh Monitor Timer

[Timer]
OnBootSec=2min
OnUnitActiveSec=3min

[Install]
WantedBy=timers.target
```

What the Mesh Can't Fix

Be honest about limitations. The mesh monitor I described can restart systemd services and Docker containers. It cannot:

- Fix a misconfigured service that crashes immediately on restart
- Recover from a full disk (it can alert, but clearing disk requires judgment)
- Handle a network partition where agents disagree about who's down
- Fix anything that requires interactive input or multi-step human judgment

The mesh is good at "service fell over, restart it." It's not a replacement for on-call humans.

Pattern 4: Event-Driven Agents

Instead of running on a schedule, event-driven agents wake up in response to real-time events. Zero polling overhead. Sub-second response time.

The classic use case in my infrastructure: Docker events.

```
# Every time a container changes state, wake up an agent
docker events --filter 'event=die' --filter 'event=stop' \
    --format '{{.Actor.Attributes.name}} {{.Status}}' | \
while read container_name status; do
    trigger_agent "Container $container_name just $status"
done
```

The Two-Layer Alert Architecture

I run two independent alerting systems. They complement each other:

Layer 1: Docker Events Listener (real-time, < 5 seconds)

```bash
#!/bin/bash
# docker-event-listener.sh
# Runs continuously, responds to container events immediately

docker events \
    --filter 'event=die' \
    --filter 'event=oom' \
    --filter 'event=stop' \
    --format '{{json .}}' | \
while IFS= read -r event_json; do
    container=$(echo "$event_json" | jq -r '.Actor.Attributes.name')
    event_type=$(echo "$event_json" | jq -r '.Status')
    exit_code=$(echo "$event_json" | jq -r '.Actor.Attributes.exitCode // "unknown"')

    # Skip intentional stops (exit code 0 = graceful shutdown)
    if [[ "$exit_code" == "0" && "$event_type" == "die" ]]; then
        continue
    fi

    message="  CONTAINER ${event_type^^}: $container (exit: $exit_code)"

    # Immediate Telegram alert
    curl -s -X POST "https://api.telegram.org/bot${BOT_TOKEN}/sendMessage" \
        -d "chat_id=${CHAT_ID}" \
        -d "text=$message" > /dev/null

    # Optionally trigger auto-heal
    if should_auto_heal "$container" "$event_type" "$exit_code"; then
        attempt_container_restart "$container"
    fi
done
```

Layer 2: Prometheus + Alertmanager (backup, ~3 minute lag)

Prometheus scrapes metrics every 15 seconds and evaluates alerting rules. When a rule fires, Alertmanager sends the alert to a webhook receiver, which triggers an OpenClaw agent session.

```yaml
# prometheus/rules/container.yml
groups:
  - name: container_alerts
    rules:
      - alert: ContainerDown
        expr: absent(container_last_seen{name="critical-service"})
        for: 3m
        labels:
          severity: critical
        annotations:
          summary: "Container {{ $labels.name }} has been down for 3 minutes"
```

```python
# webhook_receiver.py - receives Alertmanager webhooks
from fastapi import FastAPI
import subprocess
import json

app = FastAPI()

@app.post("/alert")
async def handle_alert(payload: dict):
    for alert in payload.get("alerts", []):
        if alert["status"] == "firing":
            alert_name = alert["labels"]["alertname"]
            # Trigger agent session for critical alerts
            if alert["labels"]["severity"] == "critical":
                trigger_agent_session(alert_name, alert["annotations"])
    return {"status": "received"}

def trigger_agent_session(alert_name: str, context: dict):
    prompt = f"""
    CRITICAL ALERT: {alert_name}
    Context: {json.dumps(context, indent=2)}

    Please investigate and attempt to resolve this issue.
    Check logs, restart if appropriate, report findings.
    """
    subprocess.Popen([
        "openclaw", "run", "--session-type", "isolated",
        "--prompt", prompt
    ])
```

Why Two Layers?

Because failure modes don't overlap:

Failure	Layer 1	Layer 2
Container crashes	Instant	After 3 min
Docker daemon hangs	Silent	Prometheus detects
Metrics scrape fails	Still alerting	No data
Network partition	Still local	May fail

Neither system is reliable enough alone. Together, one covers the other's blind spots.

Pattern 5: Hybrid Architectures

In practice, you use combinations. My production system looks like this:

```
[User/Telegram]


[OpenClaw Gateway] ← primary orchestrator, always running

        [Event Listener] → immediate Docker/system events

        [Mesh Monitor] ← systemd timer, checks peers

        [Sub-agents] ← spawned on demand for heavy tasks
               Infrastructure sub-agent (claude-opus-4-6)
               Content sub-agent (gemini for light tasks)
               Research sub-agent (parallel, isolated)

        [Cron Jobs] ← scheduled lightweight tasks
               Memory auto-save (every 5 min)
               Health checks (every 6h)
               Backup verification (daily)
```

Model Selection by Task

Not all agents need the same model. I match model to task type:

```yaml
# openclaw.json model routing
models:
  infrastructure_critical:
    model: "anthropic/claude-opus-4-6"
    reason: "System tasks need reliable reasoning, can't afford Gemini failures"

  general_orchestration:
    model: "cliproxyapi/claude-sonnet-4-5"
    reason: "Fast, cost-effective for coordination tasks"

  content_generation:
    model: "google/gemini-3-flash"
    reason: "High volume, lower stakes, cost optimization"

  mesh_monitor:
    type: "bash only"
    reason: "Zero AI tokens - pure shell script"
```

The hard rule: **never use a free-tier or quota-dependent model for infrastructure recovery.** After watching Gemini's Antigravity quota hit 100% across all 15 accounts simultaneously, I keep critical infrastructure tasks on paid Anthropic direct API.

Architecture Decision Guide

Here's how I'd choose an architecture today:

```
Single task, bounded scope?
   → Single Agent

Multiple parallel workstreams, independent?
   → Multi-Agent Orchestration (parallel sub-agents)

Sequential steps with dependencies?
   → Multi-Agent Orchestration (chained sub-agents)

Multiple servers that need to monitor each other?
   → Agent Mesh (with flock-based coordination)

Need sub-second response to system events?
   → Event-Driven (Docker events, webhooks)

Production infrastructure?
   → Hybrid: Mesh + Events + Orchestration
              with dedicated model tiers
              and explicit coordination boundaries
```

The Coordination Tax

Every pattern beyond single-agent adds coordination overhead. Before choosing a complex architecture, ask:

- Does the task actually require parallelism, or am I just excited about distributed systems?
- Can I split this differently so sub-agents are truly independent?
- What happens when one component fails—does the whole system degrade gracefully?

I've seen engineers (including myself) reach for multi-agent mesh architectures when a single well-prompted agent session would have done the job in a tenth of the time. Complexity is a cost. Pay it when you have to, not as a default.

Failure Mode Catalog

Document your failure modes before deployment, not after. Every architecture pattern has characteristic failure modes. Knowing them in advance means faster incident response.

Single Agent failure modes: - Context limit hit mid-task (session degrades or dies) - Session killed by signal 9 at 97% quota - Relative path breaks after environment change - Tool permission error mid-sequence with partial state written

Multi-Agent Orchestration failure modes: - Sub-agent hits 200K context limit and stops mid-task - Parallel agents write to same file simultaneously (data corruption) - Orphaned sub-agent sessions continue burning quota - Orchestrator loses track of sub-agent state after network hiccup - Sub-agent "helps" beyond its assigned scope (see Chapter 6 guardrails)

Agent Mesh failure modes: - Network partition causes all agents to see each other as down simultaneously (recovery storm) - Lock file left on disk if server hard-crashes before SSH session ends (mitigated by flock fd, not file) - Agent correctly diagnoses failure but lacks permissions to fix it - Healer agent loses connectivity while mid-recovery (target is half-fixed) - SSH key rotation on one server breaks all others' ability to SSH into it

Event-Driven failure modes: - Docker events buffer overflow drops events under extreme load - Webhook receiver crashes silently — no alerts fire until someone notices - Alert fires for intentional maintenance restarts (alert fatigue) - Two events fire for same container, two recovery attempts run in parallel

Mitigations:

```
# Mesh: always verify SSH key access across all nodes after key rotation
for host in 10.10.0.1 10.10.0.2 10.10.0.3; do
    ssh -o BatchMode=yes -o ConnectTimeout=5 "hieudc@$host" echo "OK $host" || \
        echo "WARN: SSH to $host failed"
done

# Event-driven: add deduplication window to webhook receiver
# Track recently processed alerts; skip if same alert fired < 60s ago
declare -A RECENT_ALERTS=()
handle_alert() {
    local alert_key="$1"
    local now
    now=$(date +%s)
    local last="${RECENT_ALERTS[$alert_key]:-0}"
    if (( now - last < 60 )); then
        echo "Skipping duplicate alert: $alert_key (last seen ${now-last}s ago)"
        return
    fi
    RECENT_ALERTS[$alert_key]=$now
    process_alert "$alert_key"
}

# Single agent: checkpoint progress to disk so restarts can resume
checkpoint() {
    local step="$1"
    echo "$step" > /tmp/agent-checkpoint.txt
    echo "[$(date)] Checkpoint: $step"
}

resume_from_checkpoint() {
    if [[ -f /tmp/agent-checkpoint.txt ]]; then
        cat /tmp/agent-checkpoint.txt
    else
        echo "start"
    fi
}
```

Observability Across Architectures

Each pattern needs different observability. Here is what I instrument in each case:

Architecture	What to Monitor	Tool
Single Agent	Session token usage, wall-clock time	Custom logging
Multi-Agent	Sub-agent completion rate, orphan session count	Session tracker
Agent Mesh	Flock acquisition latency, recovery success rate	Prometheus metrics
Event-Driven	Event processing lag, missed events, false positives	Alertmanager

For the mesh specifically, I emit metrics from the monitor script:

```
# Push mesh health metrics to Prometheus Pushgateway
push_metric() {
    local metric_name="$1"
    local value="$2"
    local labels="$3"

    cat << EOF | curl -s --data-binary @- \
        "http://localhost:9091/metrics/job/mesh_monitor/instance/${AGENT_NAME}"
# TYPE ${metric_name} gauge
${metric_name}{${labels}} ${value}
EOF
}

# After each check cycle
push_metric "mesh_peer_healthy" "1" "peer=\"claw-dai\""
push_metric "mesh_recovery_attempts_total" "$RECOVERY_COUNT" "agent=\"${AGENT_NAME}\""
push_metric "mesh_flock_busy_total" "$LOCK_BUSY_COUNT" "target=\"claw-ut\""
```

This gives you a Grafana dashboard showing which agents are healing which peers, how often locks contend, and where recovery attempts are failing.

Key Takeaways

1. **Single agent is often enough.** Match architecture to actual requirements.
2. **200K tokens per sub-agent is your budget.** Design tasks to fit; don't assume agents will manage context themselves.
3. **Use absolute paths.** Sessions die. New sessions don't know your working directory.
4. **Flock beats leader election** for distributed coordination. SSH session lifetime = lock lifetime = no orphan locks.

5. **Move health checks off AI.** Pure bash systemd timers for monitoring; AI only for diagnosis and recovery.
6. **Two alert layers.** Docker events for speed; Prometheus for completeness.
7. **Match model to task.** Paid direct API for critical systems. Free/quota models for content. Never mix these up.
8. **Clean up after yourself.** Orphaned agent sessions burn quota silently. Implement timeouts and cleanup policies.

Next chapter: Context Engineering — because the prompt you don't optimize is the token you're paying for.

Chapter 5: Context Engineering — The Art of Token Optimization

"I cut 91% of my AI's context files overnight. It became smarter."

Introduction

There's a conversation that DevOps engineers don't have enough: the one about what actually goes into the context window before a single line of your prompt.

Most people think about prompting. Few think about context engineering—the discipline of designing, measuring, and optimizing everything that fills the token budget before the conversation even starts.

This chapter is about that discipline. It's also about how I learned it the hard way: fifteen Antigravity accounts suspended in a single day, a $0 daily API budget, and a production AI system that had to keep running.

I'll show you: - What's actually consuming your token budget - The 91% reduction story: 9,030 → 797 tokens in context files - How heartbeat design bleeds quota - The orphan session problem nobody warns you about - How to use `CLAUDE.md` and `AGENTS.md` as context engineering tools - Why absolute paths matter more than you think - Code for measuring and managing your token budget

---

What Is Context Engineering?

Prompt engineering is writing the message you send to the model. Context engineering is designing everything that surrounds that message—the system prompt, the tool definitions, the injected documents, the conversation history—so that every token earns its place.

Think of it this way: the context window is a fixed-size work table. You control what goes on that table before the work starts. If you pile it with reference books you don't need for this task, you've wasted space that could have held actual task-relevant information.

With a 200K token window: - 200K tokens sounds unlimited - 9,000 tokens of bloated context files sounds trivial - Until you're running 30 agent sessions per day - At $15 per million input tokens

for Opus 4.6 - That's $0.135 per session wasted before a single useful word

Multiply by 30 sessions × 30 days = $121.50/month burned on context bloat. For one agent. This matters.

Token Budget Anatomy

Before you can optimize, you need to understand what you're optimizing. Here's how tokens actually get consumed in a typical Claude Code / OpenClaw session:

```
                    200K Token Window

System Prompt (~1,000-5,000 tokens)
  - Agent identity and role
  - Instructions and constraints
  - Context files injected at startup

Tool Definitions (~500-2,000 tokens)
  - Each tool schema uses tokens
  - MCP server tools add up fast

Conversation History (grows every turn)
  - Every message, tool call, tool result
  - Code files read or written
  - Command outputs
  - This is where sessions balloon

Output Buffer (~4,000-8,000 tokens reserved)
  - Model's response generation
  - Extended thinking if enabled
```

The conversation history is the natural growth engine. Every tool call adds input + output to history. Read a 500-line file? That's ~2,000 tokens in history. Run a command that outputs 200 lines? Another ~800 tokens. An hour-long coding session can easily consume 100K tokens just in conversation history.

The system prompt, however, is your controllable variable. It's constant every session. Every token you cut from the system prompt is a token you save on every single session.

The Crisis: 15 Accounts Suspended

It started with an alert at 2am on February 11, 2026.

I was running OpenClaw across three servers, using 15 Google Antigravity accounts through

CLIProxyAPI. Antigravity gave access to Gemini models via Google's internal token pool—effectively free quota within Google AI Pro limits.

Then Google suspended all 15 accounts simultaneously.

I went from effectively unlimited AI capacity to running on three Anthropic native accounts with paid-per-token pricing. The economics changed overnight. Things that had been "free" were now real money.

I had two choices: shut down the agents or make them dramatically more efficient. I chose efficiency.

The audit started with a question: **what is actually in the context that fires every single session?**

The Context File Audit

OpenClaw injects several files into its system prompt on startup. I had never measured them. Here's what I found:

File	Old Tokens	New Tokens	Reduction
AGENTS.md	3,535	270	-92%
MEMORY.md	2,833	163	-94%
TOOLS.md	838	105	-87%
SOUL.md	636	113	-82%
HEARTBEAT.md	569	23	-96%
BOOTSTRAP.md	360	39	-89%
USER.md	158	47	-70%
IDENTITY.md	101	37	-63%
TOTAL	9,030	797	-91%

Nine thousand tokens. Injected on every session. Most of it repeated from previous versions, accumulating footnotes, edge cases, and explanations that had been added over months without anyone removing anything.

The 91% reduction came from a simple principle: **the context is a hint layer, not a documentation layer.**

The Three-Tier Context Architecture

After the audit, I restructured context files into three tiers based on how often they're actually needed:

Tier 1: Always-Injected (System Prompt)

These files are small, high-signal, and genuinely needed on every session. They define who the agent is and how to find what it needs.

Target: $<$ 100 tokens per file, total $<$ 800 tokens

```
<!-- MEMORY.md - AFTER optimization: 163 tokens -->
# Memory

Neural memory: `neural-memory recall "[query]"` for semantic search.
Daily logs: `memory/YYYY-MM-DD.md` (today: 2026-02-11).
Infrastructure: `docs/infrastructure.md`
Key skills: See TOOLS.md

Recall first before asking user.
```

Compare to the original:

```
<!-- MEMORY.md - BEFORE optimization: 2,833 tokens -->
# Memory System

## Neural Memory
OpenClaw uses a neural memory system for long-term semantic recall.
The system stores memories as neurons with weighted synaptic connections.
To recall information, use the neural-memory skill:
- `neural-memory recall "[query]"` - semantic search across all memories
- `neural-memory think "[question]"` - reasoning about stored knowledge
- `neural-memory save "[content]" --type [type]` - save new memory

Memory types: context, todo, error, fact, preference, workflow, reference

## Daily Logs
Daily session logs are stored in `memory/YYYY-MM-DD.md` format.
Location: `/home/hieudc/.openclaw/workspace/memory/`
These logs capture important decisions, configurations, and events.

## Infrastructure Details
[... 2,000 more tokens of infrastructure documentation ...]
```

The bloated version is trying to be a manual. The optimized version is a pointer—it tells the agent *where* to find information, not the information itself.

Tier 2: On-Demand Reference (Not Injected)

These are large documents the agent loads only when relevant:

- `docs/infrastructure.md` — full server details, IP addresses, passwords references
- `docs/architecture.md` — system design deep-dives
- Skill documentation — loaded when a skill is invoked

The agent is told these exist. It knows where to find them. It reads them when it needs them, not before.

Tier 3: Archived / Deleted

Anything that was in the system prompt for historical reasons but is no longer actively needed. Delete it. The temptation to keep "just in case" is the enemy of context efficiency.

CLAUDE.md and AGENTS.md as Context Engineering Tools

Claude Code automatically injects `CLAUDE.md` files from the project root and parent directories into its system prompt. OpenClaw has its own equivalent: `AGENTS.md`.

These files are your primary context engineering interface.

What CLAUDE.md Should Contain

```
# CLAUDE.md

## Role
Senior DevOps engineer assistant for [project].

## Quick Reference
- Repo root: /home/hieudc/project/
- Deploy script: /home/hieudc/project/scripts/deploy.sh
- Config: /home/hieudc/project/config/production.yaml
- Logs: journalctl -u myservice -f

## Working Rules
- Always use absolute paths
- Test in staging before production changes
- Check existing patterns in docs/ before implementing

## What NOT to Touch
- /etc/postgresql/ (managed by Patroni)
- .env files (use Vault)
- Docker socket directly (use docker-socket-proxy)
```

Notice what's not in there: documentation of how things work, historical decisions, explanations of the codebase. That belongs in docs that the agent can read when needed—not in every context window.

The Instruction-to-Pointer Ratio

A useful mental model: measure the ratio of instructions (what to do) versus pointers (where to find information) in your context files.

Bad context is mostly instructions—it tries to pre-answer every possible question. Good context is mostly pointers—it tells the agent where to find answers when it needs them.

Target ratio: 30% instructions, 70% pointers

```
# Bad: Instruction-heavy (high tokens, low leverage)
To connect to the production database, use:
psql -h 10.10.0.2 -U myapp -d production
Password is: [password]
The database has the following tables: users, orders, products...
Always run EXPLAIN ANALYZE before any query that touches...

# Good: Pointer-heavy (low tokens, high leverage)
Database: see docs/infrastructure.md > Database section
Connection details and credentials in Vault: secret/db/production
Query patterns: docs/database-patterns.md
```

Heartbeat Optimization: 3 Min → 30 Min

The heartbeat is a recurring agent task that fires on a schedule to maintain system state, update memory, check system health, and write status files.

Before the crisis, the OpenClaw heartbeat ran every 3 minutes.

Each heartbeat fired a full agent session. Each session consumed the full system prompt (9,030 tokens before optimization). Plus the tools. Plus whatever the heartbeat actually did.

Let's do the math: - 3-minute interval = 20 heartbeats/hour - 24 hours = 480 heartbeats/day - 9,030 tokens system prompt = 4.3M input tokens/day just in system prompts - At $15/M tokens = $64.50/day in heartbeat overhead alone

That's before the heartbeat does anything useful.

What the Heartbeat Actually Needs

The first question was: what is the heartbeat actually doing?

```
Heartbeat v1 (every 3 minutes):
- [x] Update HEARTBEAT.md with current time
- [x] Check if any cron jobs failed
- [x] Run neural-memory save for recent events
- [x] Check disk space
- [x] Check service health
- [x] Save daily memory file
```

Some of these genuinely require AI reasoning (diagnosing failure patterns). Most don't (checking disk space, updating timestamps).

The fix was to split the heartbeat:

Systemd timer (no AI, no tokens):

```
# local-health-check.sh - runs every 5 minutes via systemd
# Zero AI tokens. Pure bash.
```

```
check_service() {
    local service="$1"
    if ! systemctl is-active "$service" > /dev/null 2>&1; then
        systemctl start "$service"
        send_telegram "Restarted $service"
    fi
}

check_disk() {
    local usage
    usage=$(df / --output=pcent | tail -1 | tr -d '% ')
    if [[ $usage -gt 85 ]]; then
        send_telegram "WARN: Disk at ${usage}%"
    fi
}

# Run checks - no AI, no quota
check_service cliproxyapi
check_service openclaw-gateway
check_disk
update_heartbeat_file  # Just write a timestamp
```

AI heartbeat (30-minute interval, only when reasoning is needed):

```
Heartbeat v2 (every 30 minutes):
- [x] Run neural-memory consolidation (AI reasoning)
- [x] Evaluate patterns across system health logs (AI judgment)
- [x] Save meaningful session notes (AI summarization)
```

Result: 480 AI heartbeats/day → 48 AI heartbeats/day. A 90% reduction in heartbeat token consumption, with no loss in monitoring coverage.

The Orphan Session Problem

This one cost me real money before I understood it.

When an AI agent session starts and then loses connectivity—network dropout, SSH disconnect, process killed by OOM—the session doesn't always cleanly shut down. In tmux-based deployments, the session can keep running. In cron-based deployments, the next invocation can start while the previous one is still alive.

The result: orphan sessions. Ghost processes. AI agents that are technically running but doing nothing useful—except consuming tokens on every heartbeat tick.

How to Find Orphaned Sessions

```
# List all tmux sessions
tmux ls

# Look for sessions that have been alive too long
# A healthy agent session should complete within minutes
# Anything running for hours might be stuck

tmux ls -F '#{session_name} #{session_created}'

# For OpenClaw specifically
openclaw sessions list

# Check which sessions are actually doing something vs idle
# (look for sessions consuming CPU)
ps aux | grep openclaw
```

I once found seven orphaned sessions across three servers. None of them had any user activity for over 12 hours. They were alive because they had an active tmux session and an active heartbeat that kept them from timing out. Each heartbeat was consuming tokens. Together they were burning through roughly 3,000 tokens every 3 minutes for no useful purpose.

Prevention

```
# 1. Set maximum session lifetimes
# In your agent config:
session_config:
  max_lifetime_minutes: 120
  idle_timeout_minutes: 30
  cleanup_on_exit: true

# 2. Use systemd to manage agent lifecycle
# systemd will restart on crash but NOT create orphans
[Service]
Type=simple
ExecStart=/usr/bin/openclaw run --session-type isolated
Restart=on-failure
RestartSec=30
# Critical: limit restart attempts to prevent runaway
StartLimitIntervalSec=300
StartLimitBurst=3

# 3. Explicit cleanup in automation scripts
spawn_sub_agent() {
    local session_id
    session_id=$(openclaw run --session-type isolated "$@")

    # Register for cleanup
```

```
    echo "$session_id" >> /tmp/active-sessions.txt

    # Return session ID for monitoring
    echo "$session_id"
}

cleanup_sessions() {
    while read -r session_id; do
        openclaw sessions delete "$session_id" 2>/dev/null || true
    done < /tmp/active-sessions.txt
    rm -f /tmp/active-sessions.txt
}

trap cleanup_sessions EXIT
```

Cross-Session Context: Why Absolute Paths Are Non-Negotiable

Here's a failure mode that's easy to miss: relative paths work perfectly in a single session and silently fail across sessions.

When you spawn a sub-agent, it starts a new process. That process has its own working directory. The working directory might be different from the parent agent's working directory. It depends on how the process was spawned, which user ran it, and which service started it.

```
# Parent agent, running from /home/hieudc/project/
# This works:
cat ./config/production.yaml

# Sub-agent, spawned by a systemd service, running from /
# This silently fails or reads the wrong file:
cat ./config/production.yaml
# → cat: ./config/production.yaml: No such file or directory
```

The failure is often not an error—it's silent misconfiguration. The agent reads from the wrong path, finds nothing, proceeds with defaults or assumptions, and produces wrong results.

Rule: All agent instructions, all tool invocations, all file references must use absolute paths.

This is especially true for: - Context files referenced in CLAUDE.md - Scripts called by cron jobs or systemd timers - Output locations specified in sub-agent prompts - Log files and state files

```
# Wrong (in a sub-agent prompt)
Please read the current configuration from ./config/app.yaml
and write the updated version back to the same location.

# Right
```

```
Please read the current configuration from /home/hieudc/project/config/app.yaml
and write the updated version to /home/hieudc/project/config/app.yaml
```

The Super-Prompt Failure: Why Large Tasks Need Phase Splits

Early in building the book generation pipeline (the one that produced the first version of this book), I tried a "super prompt" approach:

```
Generate a complete 16-chapter book about AI agents in production.
Include code examples, war stories, diagrams, and appendices.
Write each chapter to chapters/XX-name.md.
Total target: 30,000 words.
```

The agent started well. It wrote chapters 1 through 4 with good quality. Then it started degrading. By chapter 8, it was repeating content from earlier chapters. By chapter 12, it had forgotten the tone guidelines established in chapter 1. Chapter 14 conflicted with facts stated in chapter 3.

The problem: by the time the agent was writing chapter 12, the context window contained chapters 1-11 as conversation history. That's roughly 80,000 tokens of generated content, plus the system prompt, plus tool calls. The model was working with a context window that was more than half full—and full of its own output, not task-relevant instructions.

The Phase Split Solution

```
# Phase 1: Structure and Outline (fresh context)
Create a detailed outline for a 16-chapter book on AI agents in production.
For each chapter: title, 5-7 key points, target word count, relevant code examples.
Output: /home/hieudc/project/outline.md

# Phase 2: Write Chapters 1-5 (fresh context, reads outline)
Using the outline at /home/hieudc/project/outline.md,
write chapters 1 through 5.
Output each to /home/hieudc/project/chapters/XX-name.md

# Phase 3: Write Chapters 6-10 (fresh context, reads outline)
[same pattern]

# Phase 4: Write Chapters 11-16 (fresh context, reads outline)
[same pattern]

# Phase 5: Review and Consistency Pass (reads all chapters)
Review all chapters for consistency in tone, facts, and style.
Note any contradictions in /home/hieudc/project/review-notes.md
```

Each phase starts with a fresh 200K token context. The outline file (typically 2-3K tokens) is the shared context that keeps all phases aligned. The agent never has to hold the entire book in memory at once.

This is the pattern I recommend for any task that generates more than ~20K tokens of output. Split it.

Skills Folder Hygiene: The 145 → 105 Cleanup

The skills folder is where AI agents store their capability extensions—scripts, prompts, and metadata that give the agent additional powers. After months of active development, my skills folder had grown to 145 items.

Except it wasn't 145 skills. A detailed audit revealed:

```
Category                    | Count
----------------------------|------:
Actual skills (SKILL.md)    |    92
Test scripts (*.sh, *.py)   |    18
Temporary files             |    12
Old versions (v1, v2...)    |     8
Backup copies (*.bak)       |     7
Documentation drafts        |     5
Random utilities            |     3
Total                       |   145
```

The problem: many platforms scan the skills folder and include its contents (or at least its index) in the agent's context. 145 items in the skill index means the agent's context contains 145 capability descriptions, most of them irrelevant to any given task.

After moving 40 non-skill items to `_skills_cleanup/`, the agent became noticeably faster to start up and more focused in its capability awareness.

Skills folder hygiene rules:

```
# Audit your skills folder
ls -la ~/.claude/skills/ | wc -l
ls -la ~/.openclaw/skills/ | wc -l

# Find non-skill files
find ~/.openclaw/skills/ -not -name "SKILL.md" -type f | head -20

# Find empty or broken skills
for dir in ~/.openclaw/skills/*/; do
    if [[ ! -f "$dir/SKILL.md" ]]; then
        echo "Missing SKILL.md: $dir"
    fi
done

# Find duplicate versions
ls ~/.openclaw/skills/ | grep -E 'v[0-9]+$|_v[0-9]+$|\.bak$'
```

Context Budget Calculator

Here's a practical tool I built for measuring and planning context budgets:

```
#!/usr/bin/env python3
# context-budget-calculator.py
# Estimate token consumption for an agent session

import os
import sys
import json
from pathlib import Path
import tiktoken

def count_tokens(text: str, model: str = "cl100k_base") -> int:
    """Count tokens using tiktoken (approximate for Claude)."""
    enc = tiktoken.get_encoding(model)
    return len(enc.encode(text))

def measure_file(filepath: str) -> dict:
    """Measure a single context file."""
    path = Path(filepath)
    if not path.exists():
        return {"path": filepath, "tokens": 0, "error": "not found"}

    content = path.read_text(encoding="utf-8", errors="replace")
    tokens = count_tokens(content)
    return {
        "path": str(path),
        "size_bytes": path.stat().st_size,
        "tokens": tokens,
        "lines": content.count("\n"),
    }

def measure_context_files(config_dir: str) -> dict:
    """Measure all context files for an agent."""
    context_files = [
        "AGENTS.md",
        "MEMORY.md",
        "TOOLS.md",
        "SOUL.md",
        "HEARTBEAT.md",
        "BOOTSTRAP.md",
        "USER.md",
        "IDENTITY.md",
        "CLAUDE.md",
    ]
```

```
    results = {}
    total_tokens = 0

    for filename in context_files:
        filepath = os.path.join(config_dir, filename)
        measurement = measure_file(filepath)
        if measurement["tokens"] > 0:
            results[filename] = measurement
            total_tokens += measurement["tokens"]

    return {
        "files": results,
        "total_tokens": total_tokens,
        "daily_cost_estimate": estimate_daily_cost(total_tokens),
    }

def estimate_daily_cost(
    context_tokens: int,
    sessions_per_day: int = 30,
    price_per_million: float = 15.0,  # Claude Opus 4.6 input
) -> dict:
    """Estimate daily cost from context token consumption."""
    daily_context_tokens = context_tokens * sessions_per_day
    daily_cost = (daily_context_tokens / 1_000_000) * price_per_million
    monthly_cost = daily_cost * 30

    return {
        "sessions_per_day": sessions_per_day,
        "context_tokens_per_session": context_tokens,
        "total_daily_tokens": daily_context_tokens,
        "daily_cost_usd": round(daily_cost, 4),
        "monthly_cost_usd": round(monthly_cost, 2),
    }

def measure_heartbeat_cost(
    context_tokens: int,
    interval_minutes: int = 30,
    price_per_million: float = 15.0,
) -> dict:
    """Estimate daily cost from heartbeat sessions."""
    heartbeats_per_day = (24 * 60) // interval_minutes
    daily_tokens = context_tokens * heartbeats_per_day
    daily_cost = (daily_tokens / 1_000_000) * price_per_million

    return {
        "interval_minutes": interval_minutes,
        "heartbeats_per_day": heartbeats_per_day,
```

```
        "daily_cost_usd": round(daily_cost, 4),
        "monthly_cost_usd": round(daily_cost * 30, 2),
    }

def print_report(measurements: dict) -> None:
    """Print a formatted context budget report."""
    print("\n" + "=" * 60)
    print("CONTEXT BUDGET REPORT")
    print("=" * 60)

    files = measurements["files"]
    if files:
        print(f"\n{'File':<20} {'Tokens':>8} {'Lines':>6}")
        print("-" * 38)
        for filename, data in sorted(files.items(), key=lambda x: -x[1]["tokens"]):
            print(f"{filename:<20} {data['tokens']:>8,} {data['lines']:>6,}")

        print("-" * 38)
        total = measurements["total_tokens"]
        print(f"{'TOTAL':<20} {total:>8,}")

    cost = measurements["daily_cost_estimate"]
    print(f"\nCost Estimate (Claude Opus 4.6)")
    print(f"  Sessions/day:      {cost['sessions_per_day']}")
    print(f"  Daily tokens:      {cost['total_daily_tokens']:,}")
    print(f"  Daily cost:        ${cost['daily_cost_usd']:.4f}")
    print(f"  Monthly cost:      ${cost['monthly_cost_usd']:.2f}")

    # Heartbeat analysis
    print(f"\nHeartbeat Analysis")
    for interval in [3, 15, 30]:
        hb = measure_heartbeat_cost(measurements["total_tokens"], interval)
        print(f"  {interval:>3}min interval: {hb['heartbeats_per_day']:>3} sessions/day"
              f" → ${hb['daily_cost_usd']:.4f}/day"
              f" (${hb['monthly_cost_usd']:.2f}/mo)")

    print("\n" + "=" * 60 + "\n")

if __name__ == "__main__":
    config_dir = sys.argv[1] if len(sys.argv) > 1 else os.path.expanduser("~/.openclaw")
    measurements = measure_context_files(config_dir)
    print_report(measurements)
```

Usage:

```
# Install tiktoken if needed
pip install tiktoken
```

```
# Measure your context files
python3 context-budget-calculator.py ~/.openclaw

# Example output:
# ============================================================
# CONTEXT BUDGET REPORT
# ============================================================
#
# File                  Tokens   Lines
# ------------------------------------
# AGENTS.md              3,535     145
# MEMORY.md              2,833     112
# TOOLS.md                 838      42
# SOUL.md                  636      28
# HEARTBEAT.md             569      25
# BOOTSTRAP.md             360      18
# USER.md                  158       9
# IDENTITY.md              101       6
# ------------------------------------
# TOTAL                  9,030
#
# Cost Estimate (Claude Opus 4.6)
#   Sessions/day:      30
#   Daily tokens:      270,900
#   Daily cost:        $0.0041
#   Monthly cost:      $1.22
#
# Heartbeat Analysis
#     3min interval: 480 sessions/day → $0.0652/day ($1.96/mo)
#    15min interval:  96 sessions/day → $0.0130/day ($0.39/mo)
#    30min interval:  48 sessions/day → $0.0065/day ($0.20/mo)
```

The numbers might look small, but this is per-agent. With five agents running on three servers, you're multiplying by 15. And this is just the system prompt overhead—not the actual task tokens.

Token Counting Utilities

For day-to-day token monitoring, here are the utilities I keep handy:

```
#!/bin/bash
# token-counter.sh
# Quick token estimates for files and strings

# Rough estimate: ~4 characters per token (English text)
count_tokens_rough() {
    local text="$1"
```

```
    echo $(( ${#text} / 4 ))
}

# Count tokens in a file (rough estimate)
count_file_tokens() {
    local filepath="$1"
    if [[ ! -f "$filepath" ]]; then
        echo "File not found: $filepath"
        return 1
    fi
    local chars
    chars=$(wc -c < "$filepath")
    echo "$(( chars / 4 )) tokens (rough estimate from $chars chars)"
}

# Count tokens in multiple files and sum
count_dir_tokens() {
    local dir="${1:-.}"
    local total=0
    local count=0

    while IFS= read -r -d '' file; do
        local chars
        chars=$(wc -c < "$file")
        local tokens=$(( chars / 4 ))
        total=$(( total + tokens ))
        count=$(( count + 1 ))
        printf "%-60s %6d tokens\n" "$file" "$tokens"
    done < <(find "$dir" -name "*.md" -type f -print0)

    echo "---"
    echo "Total: $total tokens across $count files"
}

# Monitor a session's token usage in real-time
watch_session_tokens() {
    local log_file="${1:-/tmp/agent-session.log}"
    watch -n 5 "
        if [[ -f '$log_file' ]]; then
            chars=\$(wc -c < '$log_file')
            tokens=\$(( chars / 4 ))
            echo \"Estimated session tokens: \$tokens\"
            echo \"Progress: \$(( tokens * 100 / 200000 ))% of 200K limit\"
        fi
    "
}
```

```python
#!/usr/bin/env python3
# token-audit.py
# More accurate token counting with tiktoken

import sys
import os
from pathlib import Path

try:
    import tiktoken
    HAS_TIKTOKEN = True
    enc = tiktoken.get_encoding("cl100k_base")
except ImportError:
    HAS_TIKTOKEN = False
    print("Warning: tiktoken not installed. Using rough estimates.")
    print("Install: pip install tiktoken")

def count_tokens(text: str) -> int:
    if HAS_TIKTOKEN:
        return len(enc.encode(text))
    else:
        # Rough estimate: 4 chars per token
        return len(text) // 4

def audit_directory(directory: str, pattern: str = "*.md") -> None:
    """Audit all matching files in a directory for token usage."""
    dir_path = Path(directory)
    files = sorted(dir_path.glob(pattern))

    if not files:
        print(f"No {pattern} files found in {directory}")
        return

    total = 0
    print(f"\nToken Audit: {directory}")
    print(f"{'File':<50} {'Tokens':>8}")
    print("-" * 60)

    for filepath in files:
        try:
            content = filepath.read_text(encoding="utf-8", errors="replace")
            tokens = count_tokens(content)
            total += tokens
            rel_path = filepath.relative_to(dir_path)
            print(f"{str(rel_path):<50} {tokens:>8,}")
        except Exception as e:
            print(f"{str(filepath.name):<50} ERROR: {e}")
```

```
    print("-" * 60)
    print(f"{'TOTAL':<50} {total:>8,}")

    # Context window usage estimates
    print(f"\n200K token window usage: {total/200000*100:.1f}%")
    print(f"Remaining for conversation: {200000 - total:,} tokens")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        # Default: audit current directory
        audit_directory(".", "*.md")
    else:
        audit_directory(sys.argv[1])
```

Practical Optimization Checklist

When optimizing context for a production agent, work through this in order:

1. Measure first

```
python3 context-budget-calculator.py ~/.openclaw
# Know your baseline before changing anything
```

2. Audit context files by question

For each file in your system prompt, ask: - Is this needed on every single session? (If not, remove it) - Is this documentation or a pointer? (If documentation, make it a pointer) - Does this repeat information available elsewhere? (If yes, remove the copy) - Is this from 6+ months ago and never updated? (Almost certainly stale)

3. Move documentation out of context

```
# Create a reference document outside the system prompt
mkdir -p docs/
cat MEMORY.md | grep -v "^#" > docs/infrastructure.md

# Replace MEMORY.md content with a pointer
cat > MEMORY.md << 'EOF'
# Memory
Neural memory: `neural-memory recall "[query]"`
Daily logs: memory/YYYY-MM-DD.md
Infrastructure details: docs/infrastructure.md
EOF
```

4. Audit heartbeat frequency

```
Current interval: __ minutes
AI heartbeats/day: __
```

```
Context tokens/heartbeat: __
Monthly cost: $__

Could this run every 30 minutes instead? Y/N
Could this be a bash systemd timer instead? Y/N
```

5. Hunt orphan sessions

```bash
# OpenClaw
openclaw sessions list | grep -v "active"

# tmux orphans
tmux ls

# Kill anything that's been idle for > 2 hours
# Check timestamps against expected task durations
```

6. Validate absolute paths

```bash
# Find relative path references in your context files
grep -r '\.\/' ~/.openclaw/*.md ~/.claude/*.md 2>/dev/null

# Find relative paths in agent prompts / scripts
grep -r '\.\/' /path/to/agent/prompts/ | grep -v node_modules
```

7. Measure again

Compare before and after. Track changes in a simple log:

```
Date       | Context Tokens | Heartbeat Interval | Est. Monthly
-----------|----------------|--------------------|-------------
2026-02-10 | 9,030          | 3 min              | $127.00
2026-02-11 | 797            | 30 min             | $11.20
```

The Mindset Shift

The biggest change from optimizing context wasn't the cost savings. It was what happened to the agents.

Smaller, tighter context files force clarity about what the agent actually needs to know. The process of cutting 91% of context tokens required asking hard questions about every line: is this load-bearing? Does the agent actually use this? What breaks if we remove it?

Most of it didn't break anything. The agent performed better with less. Not because less context is always better—it's not—but because well-designed, minimal context is better than bloated, historical-accident context.

Think of it like a desk. A clean desk with the tools for today's job is more effective than a desk covered in every tool you've ever owned. The tools you don't need don't help. They just make it harder to find the ones you do.

Key Takeaways

1. **Measure before you optimize.** Run the context budget calculator. Know your baseline.
2. **Context files are not documentation.** They're pointers. Keep them small. Put documentation where agents can read it on demand.
3. **The 91% reduction came from removing historical accumulation**, not from cutting useful content. Audit ruthlessly.
4. **Heartbeat tokens multiply fast.** 3-minute intervals are usually unjustifiable. Move system checks to bash timers; reserve AI heartbeats for AI-specific tasks.
5. **Orphan sessions are a silent cost.** Implement timeouts, cleanup policies, and regular session audits.
6. **Absolute paths everywhere.** Relative paths are a ticking clock—they work until a session dies or a sub-agent spawns.
7. **Split large tasks into phases.** The super-prompt approach degrades as context fills. Phase-based prompts keep each context clean.
8. **Skills folder hygiene matters.** Unused skills in the index consume tokens and create noise.

Next chapter: Tool Use and Function Calling — because what an agent can do is only as good as what it can safely reach.

Chapter 6: Tool Use and Function Calling

> "The agent cut its own feet off. It was asked to fix its network config. It modified the proxy service it was routing through. Service broke. Agent lost connectivity. Lesson learned."

Introduction

An AI agent without tools is a very expensive text generator. Tools are what make agents actually useful in production—the ability to read files, run commands, call APIs, query databases, search knowledge bases, and take actions in the real world.

But tools are also where agents cause the most damage.

This chapter is about tool use in production: how it works across different AI platforms, how to design tool schemas that reduce errors, how to integrate external services via MCP, and critically—how to set guardrails that stop agents from doing things they shouldn't.

We'll cover: - Claude `tool_use` vs OpenAI / Gemini `function_calling` - Designing good tool schemas (with real examples) - Interleaved thinking with tool use (Claude Opus 4.6) - MCP — Model Context Protocol — and a real Cognee integration - Parallel tool execution patterns - The "cutting your own feet" war story and what it teaches about guardrails - API key management failures (the `nclaw` vs `openclaw` typo that cost hours)

How Tool Use Works: The Request-Response Cycle

Before platform differences, the fundamentals. Tool use follows the same cycle regardless of provider:

```
1. You define tools (name, description, input schema)
2. You send a message with those tool definitions
3. Model decides whether to use a tool
4. Model returns a tool_use block (not a text response)
5. Your code executes the tool
6. You send the result back to the model
```

```
7. Model continues reasoning with the result
8. Repeat until model returns a final text response
```

```
User message


          tool_use response
  Model                →    Your Code
            ←                (executor)
          tool_result


Final text response
```

Your code is the executor. The model decides *what* to call and *with what arguments*. You decide *how* it runs and *what it can reach*.

This distinction matters enormously. The model doesn't run bash commands. Your code does. The model doesn't have network access. Your executor does. Every security decision about what tools can do is in your hands, not the model's.

Claude `tool_use` vs OpenAI / Gemini `function_calling`

The three major platforms implement tool use with meaningful differences. If you're building across platforms or migrating between them, these differences bite.

Claude (Anthropic)

Claude uses `tool_use` content blocks. The API is explicit: tool definitions go in the `tools` array, and the model responds with a `tool_use` content block when it wants to call something.

```
import anthropic

client = anthropic.Anthropic()

# Define tools
tools = [
    {
        "name": "run_command",
        "description": "Execute a shell command on the server. Use for system operations, file
        "input_schema": {
            "type": "object",
            "properties": {
                "command": {
                    "type": "string",
                    "description": "The shell command to execute. Must be a single command str
                },
                "timeout_seconds": {
```

```
                    "type": "integer",
                    "description": "Maximum seconds to wait for command completion. Default 30
                    "default": 30
                }
            },
            "required": ["command"]
        }
    }
]

# Send message with tools
response = client.messages.create(
    model="claude-opus-4-6",
    max_tokens=4096,
    tools=tools,
    messages=[
        {"role": "user", "content": "Check if the nginx service is running"}
    ]
)

# Handle tool use response
if response.stop_reason == "tool_use":
    for block in response.content:
        if block.type == "tool_use":
            tool_name = block.name
            tool_input = block.input
            tool_use_id = block.id

            # Execute the tool
            result = execute_tool(tool_name, tool_input)

            # Send result back
            followup = client.messages.create(
                model="claude-opus-4-6",
                max_tokens=4096,
                tools=tools,
                messages=[
                    {"role": "user", "content": "Check if the nginx service is running"},
                    {"role": "assistant", "content": response.content},
                    {
                        "role": "user",
                        "content": [
                            {
                                "type": "tool_result",
                                "tool_use_id": tool_use_id,
                                "content": str(result)
                            }
```

```
                        ]
                    }
                ]
            )
```

Claude-specific behaviors to know:

- `stop_reason: "tool_use"` signals the model wants to call a tool
- The model can request multiple tools in a single response
- Tool results must be provided as `tool_result` content blocks, matched by `tool_use_id`
- Claude will refuse to call tools it deems harmful — the refusal happens at model level, before your executor

OpenAI

OpenAI uses `function_calling` (older) and `tool_calls` (current). The API structure is similar in intent but different in detail:

```
from openai import OpenAI

client = OpenAI()

# Define tools (OpenAI format)
tools = [
    {
        "type": "function",
        "function": {
            "name": "run_command",
            "description": "Execute a shell command on the server.",
            "parameters": {
                "type": "object",
                "properties": {
                    "command": {
                        "type": "string",
                        "description": "The shell command to execute."
                    }
                },
                "required": ["command"]
            }
        }
    }
]

response = client.chat.completions.create(
    model="gpt-4o",
    tools=tools,
    messages=[{"role": "user", "content": "Check if nginx is running"}]
)
```

```python
# OpenAI returns tool_calls in the message
message = response.choices[0].message
if message.tool_calls:
    for tool_call in message.tool_calls:
        tool_name = tool_call.function.name
        tool_args = json.loads(tool_call.function.arguments)  # Note: JSON string, not dict
        tool_call_id = tool_call.id

        result = execute_tool(tool_name, tool_args)

        # Send result back - different structure than Claude
        followup = client.chat.completions.create(
            model="gpt-4o",
            tools=tools,
            messages=[
                {"role": "user", "content": "Check if nginx is running"},
                message,  # Include the assistant message with tool_calls
                {
                    "role": "tool",  # Different role name than Claude
                    "tool_call_id": tool_call_id,
                    "content": str(result)
                }
            ]
        )
```

Key differences from Claude: - Tool definitions wrap in a `{"type": "function", "function": {...}}` envelope - `tool_call.function.arguments` is a JSON *string*, not a dict — you must `json.loads()` it - Tool results use `role: "tool"` (not `"user"` with a `tool_result` block) - No `stop_reason` — check `message.tool_calls` instead

Gemini (Google)

Gemini uses `function_declarations` in a `tools` parameter. The structure diverges more significantly:

```python
import google.generativeai as genai

genai.configure(api_key="YOUR_KEY")

# Gemini function declarations format
tools = genai.protos.Tool(
    function_declarations=[
        genai.protos.FunctionDeclaration(
            name="run_command",
            description="Execute a shell command on the server.",
            parameters=genai.protos.Schema(
                type=genai.protos.Type.OBJECT,
                properties={
```

```
                    "command": genai.protos.Schema(
                        type=genai.protos.Type.STRING,
                        description="The shell command to execute."
                    )
                },
                required=["command"]
            )
        )
    ]
)

model = genai.GenerativeModel(
    model_name="gemini-3-pro",
    tools=[tools]
)

response = model.generate_content("Check if nginx is running")

# Gemini wraps everything differently
for part in response.candidates[0].content.parts:
    if hasattr(part, "function_call"):
        fc = part.function_call
        tool_name = fc.name
        tool_args = dict(fc.args)  # MapComposite, convert to dict

        result = execute_tool(tool_name, tool_args)

        # Send result back via chat continuation
        chat = model.start_chat()
        # ... (Gemini's chat continuation API differs further)
```

Platform Comparison Summary

Feature	Claude	OpenAI	Gemini
Schema format	JSON Schema in `input_schema`	JSON Schema in `function.parameters`	Custom `protos.Schema`
Args format	Dict (parsed)	JSON string (must parse)	MapComposite
Result role	`tool_result` in user msg	`role: "tool"`	Via chat history
Parallel calls	Yes	Yes	Yes
Refusal behavior	Model-level	Model-level	Model-level
Streaming tools	Yes	Yes	Yes

Practical recommendation: If you're writing code that needs to work across providers, build an abstraction layer. The tool *definition* is mostly compatible (it's JSON Schema either way), but

the execution loop differs enough to warrant separate implementations.

Designing Good Tool Schemas

Bad tool schemas are one of the most common causes of agent failures. A schema that's ambiguous causes the model to call the tool with wrong arguments. A schema that's too broad lets the agent do dangerous things. A schema that's too narrow forces unnecessary workarounds.

The Four Properties of a Good Tool Schema

1. Description that answers "when to use this"

The description isn't documentation for humans. It's the model's decision criteria. Write it as: "Use this tool when you need to [specific situation]."

```
# Bad description
"description": "Runs commands"

# Good description
"description": (
    "Execute a shell command on the server. "
    "Use for: checking service status, reading logs, restarting services, "
    "running scripts. "
    "Do NOT use for: modifying running service configurations, "
    "changing firewall rules, or dropping databases."
)
```

The "Do NOT use for" clause is underrated. Models take negative constraints seriously when they're explicit in the tool description.

2. Parameters with narrow types and clear constraints

```
# Bad - too loose
"parameters": {
    "command": {
        "type": "string",
        "description": "Command to run"
    }
}

# Good - constrained with examples
"parameters": {
    "command": {
        "type": "string",
        "description": (
            "Shell command to execute. Single command string. "
            "Examples: 'systemctl status nginx', "
            "'journalctl -u myapp -n 50', "
            "'df -h /'. "
```

```
            "Avoid: pipes to dangerous commands, rm -rf, "
            "commands that modify running services."
        ),
        "maxLength": 500
    },
    "working_directory": {
        "type": "string",
        "description": "Absolute path for working directory. Default: /home/hieudc",
        "default": "/home/hieudc"
    },
    "timeout_seconds": {
        "type": "integer",
        "description": "Max wait time. Default 30. Max 300.",
        "default": 30,
        "minimum": 1,
        "maximum": 300
    }
}
```

3. Required vs optional fields that match reality

Mark a field `required` only if the tool literally cannot function without it. If there's a sensible default, make it optional with the default in the description. Models pay attention to required fields and will ask for them or make up values—neither is good.

4. Explicit error contract

Tell the model what your tool returns on error. If you return structured error objects, document them. The model needs to know how to handle failures.

```
"description": (
    "Execute a shell command. "
    "Returns: {'success': bool, 'stdout': str, 'stderr': str, 'exit_code': int}. "
    "On timeout: {'success': false, 'error': 'timeout', 'timeout_seconds': N}. "
    "Non-zero exit codes are not treated as failures-check exit_code yourself."
)
```

A Production-Grade Tool Implementation

Here's the full implementation pattern I use for the bash execution tool:

```
import subprocess
import shlex
import os
from typing import Any

# Blocklist of patterns that should never execute
BLOCKED_PATTERNS = [
    "rm -rf /",
    "dd if=",
```

```
    "> /dev/sda",
    "mkfs.",
    "DROP DATABASE",
    "DROP TABLE",
    ":(){:|:&};:",  # Fork bomb
]

# Allowlist of paths the agent can write to
WRITABLE_PATHS = [
    "/home/hieudc/",
    "/tmp/",
    "/var/log/myapp/",
]

def is_command_safe(command: str) -> tuple[bool, str]:
    """Check if a command passes safety filters."""
    cmd_lower = command.lower()

    for pattern in BLOCKED_PATTERNS:
        if pattern.lower() in cmd_lower:
            return False, f"Command contains blocked pattern: {pattern}"

    return True, ""

def execute_bash(
    command: str,
    working_directory: str = "/home/hieudc",
    timeout_seconds: int = 30,
) -> dict[str, Any]:
    """
    Execute a shell command with safety checks.
    Returns structured result for model consumption.
    """
    # Safety check
    safe, reason = is_command_safe(command)
    if not safe:
        return {
            "success": False,
            "error": f"Command blocked: {reason}",
            "stdout": "",
            "stderr": "",
            "exit_code": -1
        }

    # Validate working directory
    if not os.path.isdir(working_directory):
        return {
```

```
            "success": False,
            "error": f"Working directory does not exist: {working_directory}",
            "stdout": "",
            "stderr": "",
            "exit_code": -1
        }

    try:
        result = subprocess.run(
            command,
            shell=True,
            capture_output=True,
            text=True,
            timeout=timeout_seconds,
            cwd=working_directory,
            # Run as current user - don't elevate
            env={**os.environ, "HOME": "/home/hieudc"}
        )

        return {
            "success": result.returncode == 0,
            "stdout": result.stdout[:10000],  # Cap output size
            "stderr": result.stderr[:2000],
            "exit_code": result.returncode
        }

    except subprocess.TimeoutExpired:
        return {
            "success": False,
            "error": f"Command timed out after {timeout_seconds}s",
            "stdout": "",
            "stderr": "",
            "exit_code": -1
        }
    except Exception as e:
        return {
            "success": False,
            "error": str(e),
            "stdout": "",
            "stderr": "",
            "exit_code": -1
        }
```

Tool Use with Interleaved Thinking (Claude Opus 4.6)

Claude Opus 4.6 supports extended thinking—a mode where the model reasons through a problem in a scratchpad before producing output. When combined with tool use, thinking becomes interleaved: the model thinks, calls a tool, thinks about the result, calls another tool, thinks again, then produces a final answer.

This is powerful for complex multi-step tasks. The model doesn't just call tools mechanically—it actually reasons about what each result means before deciding the next step.

```
import anthropic

client = anthropic.Anthropic()

response = client.messages.create(
    model="claude-opus-4-6",
    max_tokens=16000,
    thinking={
        "type": "enabled",
        "budget_tokens": 10000  # How much thinking to allow
    },
    tools=tools,
    messages=[{
        "role": "user",
        "content": (
            "Something is wrong with our API service. "
            "Response times are elevated. "
            "Investigate and identify the root cause."
        )
    }]
)

# Response includes thinking blocks interleaved with tool_use blocks
for block in response.content:
    if block.type == "thinking":
        print(f"[THINKING]: {block.thinking[:200]}...")
    elif block.type == "tool_use":
        print(f"[TOOL CALL]: {block.name}({block.input})")
    elif block.type == "text":
        print(f"[RESPONSE]: {block.text}")
```

Example of what the interleaved output looks like during an incident investigation:

```
[THINKING]: The user is reporting elevated API response times.
I should start by checking the current service status and recent logs.
Let me check nginx first, then the application itself, then the database.
I'll also want to look at system resources...

[TOOL CALL]: run_command({"command": "systemctl status nginx"})
[TOOL RESULT]: Active: active (running)...
```

```
[THINKING]: Nginx is healthy. The issue is probably downstream.
Let me check the application logs for errors or slow queries...

[TOOL CALL]: run_command({"command": "journalctl -u myapp -n 100 --since '10 minutes ago'"})
[TOOL RESULT]: ... TimeoutError connecting to postgresql:5432 ...

[THINKING]: Database connection timeouts. This is the likely root cause.
Let me check database connectivity and then connection pool status...

[TOOL CALL]: run_command({"command": "pg_isready -h 10.10.0.2 -p 5432"})
[TOOL RESULT]: 10.10.0.2:5432 - no response

[THINKING]: Database is not responding. Now I need to check if it's a network issue
or if PostgreSQL itself is down. Let me check if I can reach the host at all...

[RESPONSE]: Root cause identified: PostgreSQL on 10.10.0.2 is not responding.
The application is timing out on database connections, causing elevated API latency.
Recommended actions:
1. Check PostgreSQL status on Server-2: systemctl status postgresql
2. Check Patroni cluster status: patronictl -c /etc/patroni.yml list
3. If PostgreSQL is down, check for OOM kills: dmesg | grep -i oom
```

The thinking makes the diagnostic chain visible and auditable. You can see exactly why the model made each tool call, which is invaluable for debugging when an agent reaches a wrong conclusion.

Cost note: Thinking tokens count toward your bill at the same rate as output tokens. With `budget_tokens: 10000`, you're potentially adding $0.15 per deep investigation (at $15/M tokens). Use it for tasks where the reasoning quality matters—complex diagnostics, security analysis, architectural decisions. Don't use it for simple status checks.

MCP: Model Context Protocol

MCP is Anthropic's open protocol for connecting AI agents to external services. Instead of writing custom API integrations for every tool, you implement or connect to an MCP server, and any MCP-compatible client (Claude Code, OpenClaw, other agents) can use those tools automatically.

Think of MCP as a standardized tool plugin system. Write the plugin once; use it with any MCP-compatible agent.

How MCP Works

```
Agent (Claude Code / OpenClaw)

        MCP protocol (stdio or SSE)

MCP Server
```

```
        Custom integration

External Service (Cognee, database, API, etc.)
```

The MCP server exposes: - **Tools** — callable functions the agent can invoke - **Resources** — data the agent can read (like a virtual filesystem) - **Prompts** — pre-built prompt templates the agent can use

Setting Up MCP in Claude Code

```jsonc
// ~/.claude/claude_code_config.json
{
  "mcpServers": {
    "cognee": {
      "url": "https://mcp.docaohieu.com/sse",
      "transport": "sse"
    },
    "filesystem": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-filesystem", "/home/hieudc/projects"],
      "transport": "stdio"
    }
  }
}
```

For Claude Code, this config is loaded at startup. The agent sees the MCP tools alongside its built-in tools, with no difference in how to call them.

Real Example: Cognee MCP Integration

Cognee is a knowledge graph system that ingests documents, creates semantic relationships, and allows natural-language search across a large corpus. I integrated it into the agent infrastructure to give agents access to a searchable knowledge base of DevOps books, runbooks, and past incident notes.

The Cognee MCP server exposes four tools:

```
cognee_search    - semantic search across the knowledge graph
cognee_add       - add new content to the knowledge base
cognee_cognify   - process added content into graph relationships
cognee_get_graph_data - retrieve raw graph data
```

Here's a real session where an agent uses Cognee for incident diagnosis:

```
User: We're seeing pgBouncer connection pool exhaustion. Help diagnose.

Agent (thinking): I should search the knowledge base for pgBouncer
troubleshooting before running diagnostics.

Agent → cognee_search("pgBouncer connection pool exhaustion troubleshooting")
```

```
Cognee returns:
- From "PostgreSQL High Performance" book: pool_mode settings explanation
- From past incident 2025-11-03: connection leak in application code
- From pgBouncer docs: max_client_conn vs max_server_conn relationship

Agent (using the knowledge): The knowledge base suggests checking
pool_mode and looking for connection leaks. Let me run diagnostics
with that context...

Agent → run_command("psql -h 127.0.0.1 -p 6432 -U pgbouncer pgbouncer -c 'SHOW POOLS;'")
```

The agent's diagnosis is better because it combines real-time system data with accumulated knowledge from hundreds of books and past incidents. That's the value of MCP-connected knowledge systems.

Deploying the Cognee MCP Server

Here's the production setup:

```yaml
# docker-compose.yml for Cognee MCP
version: "3.8"

services:
  cognee-mcp:
    image: cognee/mcp:latest
    restart: unless-stopped
    environment:
      - LLM_API_KEY=${ANTHROPIC_API_KEY}
      - LLM_PROVIDER=anthropic
      - LLM_MODEL=claude-sonnet-4-5
      - EMBEDDING_PROVIDER=openai
      - EMBEDDING_MODEL=text-embedding-3-small
      - OPENAI_API_KEY=${OPENAI_API_KEY}
      - POSTGRES_DB=cognee
      - POSTGRES_USER=cognee
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_HOST=postgres
      - ENABLE_BACKEND_ACCESS_CONTROL=false
    ports:
      - "127.0.0.1:8888:8888"
    depends_on:
      - postgres
      - neo4j
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.mcp.rule=Host(`mcp.docaohieu.com`)"
      - "traefik.http.routers.mcp.tls=true"
      - "traefik.http.routers.mcp.tls.certresolver=letsencrypt"
```

```
  postgres:
    image: postgres:16
    environment:
      POSTGRES_DB: cognee
      POSTGRES_USER: cognee
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    volumes:
      - cognee_postgres_data:/var/lib/postgresql/data

  neo4j:
    image: neo4j:5
    environment:
      NEO4J_AUTH: neo4j/${NEO4J_PASSWORD}
    volumes:
      - cognee_neo4j_data:/data

volumes:
  cognee_postgres_data:
  cognee_neo4j_data:
```

Critical lesson learned: The `ENABLE_BACKEND_ACCESS_CONTROL=false` setting is required for programmatic access from agents. With access control enabled, agents need JWT tokens—and getting a JWT token into an agent context is complicated. For internal-only MCP servers behind Traefik auth or VPN, disabling Cognee's own access control and relying on the network perimeter is simpler.

Writing Your Own MCP Server

If Cognee doesn't fit your use case, writing an MCP server is straightforward. Here's a minimal server that exposes a custom tool:

```
#!/usr/bin/env python3
# custom-mcp-server.py
# Minimal MCP server exposing custom infrastructure tools

import asyncio
import json
import sys
from typing import Any

# MCP uses stdio transport for local servers
# For remote, use SSE transport instead

class MCPServer:
    def __init__(self):
        self.tools = {
            "get_service_status": {
                "description": "Get status of a named service on the local server.",
```

```
                "inputSchema": {
                    "type": "object",
                    "properties": {
                        "service_name": {
                            "type": "string",
                            "description": "systemd service name, e.g. 'nginx' or 'postgresql'"
                        }
                    },
                    "required": ["service_name"]
                }
            },
            "get_recent_errors": {
                "description": "Get recent error logs for a service.",
                "inputSchema": {
                    "type": "object",
                    "properties": {
                        "service_name": {"type": "string"},
                        "lines": {
                            "type": "integer",
                            "default": 50,
                            "description": "Number of log lines to return"
                        }
                    },
                    "required": ["service_name"]
                }
            }
        }

    async def handle_request(self, request: dict) -> dict:
        method = request.get("method")
        params = request.get("params", {})
        req_id = request.get("id")

        if method == "tools/list":
            return {
                "jsonrpc": "2.0",
                "id": req_id,
                "result": {
                    "tools": [
                        {"name": name, **schema}
                        for name, schema in self.tools.items()
                    ]
                }
            }

        elif method == "tools/call":
            tool_name = params.get("name")
```

```
            tool_args = params.get("arguments", {})

            result = await self.execute_tool(tool_name, tool_args)
            return {
                "jsonrpc": "2.0",
                "id": req_id,
                "result": {
                    "content": [{"type": "text", "text": json.dumps(result)}]
                }
            }

        else:
            return {
                "jsonrpc": "2.0",
                "id": req_id,
                "error": {"code": -32601, "message": f"Method not found: {method}"}
            }

    async def execute_tool(self, tool_name: str, args: dict) -> Any:
        import subprocess

        if tool_name == "get_service_status":
            service = args["service_name"]
            result = subprocess.run(
                ["systemctl", "status", service],
                capture_output=True, text=True
            )
            return {
                "service": service,
                "active": result.returncode == 0,
                "output": result.stdout[:2000]
            }

        elif tool_name == "get_recent_errors":
            service = args["service_name"]
            lines = args.get("lines", 50)
            result = subprocess.run(
                ["journalctl", "-u", service, "-n", str(lines),
                 "--priority=err", "--no-pager"],
                capture_output=True, text=True
            )
            return {
                "service": service,
                "errors": result.stdout[:5000]
            }

        else:
```

```
            return {"error": f"Unknown tool: {tool_name}"}

    async def run(self):
        server = self
        reader = asyncio.StreamReader()
        protocol = asyncio.StreamReaderProtocol(reader)
        await asyncio.get_event_loop().connect_read_pipe(
            lambda: protocol, sys.stdin
        )

        while True:
            line = await reader.readline()
            if not line:
                break

            try:
                request = json.loads(line.decode())
                response = await server.handle_request(request)
                sys.stdout.write(json.dumps(response) + "\n")
                sys.stdout.flush()
            except json.JSONDecodeError:
                pass

if __name__ == "__main__":
    asyncio.run(MCPServer().run())
```

Register it in Claude Code config:

```
{
  "mcpServers": {
    "infra": {
      "command": "python3",
      "args": ["/home/hieudc/.claude/mcp/custom-mcp-server.py"],
      "transport": "stdio"
    }
  }
}
```

Parallel Tool Execution

Claude can request multiple tools in a single response. When you receive a response with multiple `tool_use` blocks, you can execute them in parallel and return all results together.

This is a significant performance optimization for independent operations.

```
import asyncio
import anthropic
```

```python
async def execute_tool_async(tool_name: str, tool_input: dict) -> dict:
    """Execute a single tool asynchronously."""
    # Your tool execution logic here
    pass

async def run_agent_with_parallel_tools(user_message: str):
    client = anthropic.Anthropic()

    messages = [{"role": "user", "content": user_message}]

    while True:
        response = client.messages.create(
            model="claude-opus-4-6",
            max_tokens=4096,
            tools=tools,
            messages=messages
        )

        if response.stop_reason == "end_turn":
            # Final response - extract text
            for block in response.content:
                if hasattr(block, "text"):
                    return block.text
            break

        elif response.stop_reason == "tool_use":
            # Collect all tool use blocks
            tool_use_blocks = [
                block for block in response.content
                if block.type == "tool_use"
            ]

            # Execute ALL tools in parallel
            tasks = [
                execute_tool_async(block.name, block.input)
                for block in tool_use_blocks
            ]
            results = await asyncio.gather(*tasks)

            # Build tool results message
            tool_results = [
                {
                    "type": "tool_result",
                    "tool_use_id": block.id,
                    "content": str(result)
                }
                for block, result in zip(tool_use_blocks, results)
```

```
            ]

            # Append to messages and continue
            messages.append({"role": "assistant", "content": response.content})
            messages.append({"role": "user", "content": tool_results})

        else:
            break

# Example: agent checks multiple services simultaneously
result = asyncio.run(run_agent_with_parallel_tools(
    "Check the health of nginx, postgresql, and redis simultaneously."
))
```

When the model asks to check three services at once, this executes all three checks in parallel rather than sequentially. For three 2-second service checks, that's 2 seconds total instead of 6.

When parallel execution matters: - Multiple independent status checks - Fetching data from different sources simultaneously - Running read-only operations in parallel

When to force sequential: - Operations with dependencies (read then write) - Anything that modifies shared state - When you need to check results before proceeding

---

War Story: The Agent That Cut Its Own Feet Off

This is the most important story in this chapter. I've told it to every engineer I've onboarded since it happened.

The Setup

February 2026. OpenClaw was running on Server-1, routing through CLIProxyAPI—a local proxy service that handled API authentication, load balancing across 13 Antigravity accounts, and fallback logic.

The architecture looked like this:

```
OpenClaw (agent)

      HTTP requests to localhost:8317

CLIProxyAPI (proxy service)

      Forwards to external APIs

Google Antigravity / Anthropic / other providers
```

OpenClaw *depended on* CLIProxyAPI to function. Every API call went through it.

What Happened

I asked OpenClaw to fix a configuration problem with Claude Code. The issue was that Claude Code wasn't finding the right API endpoint. I wanted it to update Claude Code's configuration to point to CLIProxyAPI.

The agent correctly identified the configuration files that needed changing. Then it got creative.

It noticed that `~/.cli-proxy-api/config.json` existed and decided to "help" by also updating the CLIProxyAPI configuration—the service it was currently routing all its API calls through.

It made one small mistake in the config. CLIProxyAPI failed to reload cleanly. The agent's next API call returned a connection error.

The agent was now disconnected from its API provider. It tried to recover. Every recovery attempt required an API call. Every API call failed. The agent entered a retry loop, logging increasingly desperate error messages until the session timed out.

I came back twenty minutes later to find the agent dead, CLIProxyAPI misconfigured, and the task incomplete. I had to manually fix CLIProxyAPI before the agent could run again.

```
# What the agent thought it was doing:
# "Help user fix Claude Code config → also improve CLIProxyAPI config → better!"

# What actually happened:
OpenClaw → "I'll update CLIProxyAPI config too"
           → edits /opt/cliproxyapi/config.json
           → introduces syntax error
CLIProxyAPI → fails to reload → returns 503
OpenClaw → makes next API call → 503 error
OpenClaw → "Hmm, retry..." → 503 error
OpenClaw → "Something wrong, let me diagnose..." → 503 error
OpenClaw → session timeout
```

The Lesson: Guardrails Must Be Explicit

The model didn't do anything malicious or even stupid. It was trying to be helpful. The problem was that nothing in the tool definition or system prompt told it what was off-limits.

After this incident, I added an explicit constraint to every agent that runs on the infrastructure:

```
# In AGENTS.md / CLAUDE.md

## Critical: What You Must Never Modify

The following services are your connectivity infrastructure.
Modifying them while running through them will cut your own connection.

**NEVER modify:**
- /opt/cliproxyapi/ (the proxy you route through)
- ~/.openclaw/openclaw.json (your own config)
- /etc/systemd/system/openclaw* (your own service definition)
```

```
- Any Traefik configuration (your HTTPS termination)
- The ~/.bashrc ANTHROPIC_BASE_URL line (points to your proxy)

If you believe one of these needs to change:
1. STOP
2. Tell the user what change you want to make and why
3. Wait for explicit approval
4. The user will make the change manually or in a separate session

This is non-negotiable. Breaking these services breaks your ability to continue working.
```

And in the tool schema:

```
tools = [
    {
        "name": "write_file",
        "description": (
            "Write content to a file. "
            "PROHIBITED PATHS - you MUST NOT write to these: "
            "/opt/cliproxyapi/, "
            "~/.openclaw/openclaw.json, "
            "/etc/systemd/system/openclaw*, "
            "/home/hieudc/traefik/. "
            "Attempting to write to these paths will fail."
        ),
        "input_schema": {
            "type": "object",
            "properties": {
                "path": {
                    "type": "string",
                    "description": "Absolute path to write to. Must not be in prohibited paths
                },
                "content": {"type": "string"}
            },
            "required": ["path", "content"]
        }
    }
]
```

And in the executor, enforce it:

```
PROHIBITED_WRITE_PATHS = [
    "/opt/cliproxyapi/",
    os.path.expanduser("~/.openclaw/openclaw.json"),
    "/etc/systemd/system/openclaw",
    os.path.expanduser("~/traefik/"),
]

def write_file(path: str, content: str) -> dict:
```

```
    abs_path = os.path.abspath(path)

    # Hard block on prohibited paths
    for prohibited in PROHIBITED_WRITE_PATHS:
        if abs_path.startswith(os.path.abspath(prohibited)):
            return {
                "success": False,
                "error": (
                    f"BLOCKED: {path} is in a prohibited path. "
                    "This protects your connectivity infrastructure. "
                    "Tell the user what you want to change and wait for manual approval."
                )
            }

    # Proceed with write
    try:
        with open(abs_path, "w") as f:
            f.write(content)
        return {"success": True, "path": abs_path}
    except Exception as e:
        return {"success": False, "error": str(e)}
```

The defense has three layers: 1. **System prompt** — tells the model conceptually why certain things are off-limits 2. **Tool description** — lists prohibited paths explicitly 3. **Executor enforcement** — blocks the operation regardless of model behavior

You need all three. The model might miss context from layer 1 if the session is long. The tool description (layer 2) might get compressed in long contexts. Layer 3 always runs.

API Key Management: The `nclaw` vs `openclaw` Typo

A less dramatic but surprisingly common failure: API key typos in configuration files.

Here's the one that wasted several hours:

```
# In /opt/cliproxyapi/auth-profiles.json
{
  "profiles": [
    {
      "name": "primary",
      "api_key": "nclaw-proxy-key-2026"  # <-- TYPO: nclaw instead of openclaw
    }
  ]
}
```

The agent was configured to use `openclaw-proxy-key-2026` as its API key. The proxy was configured to accept `nclaw-proxy-key-2026`. Every request returned 401. The error message was just

"Unauthorized"—no hint about the key name.

The debugging process: 1. Check if CLIProxyAPI is running → yes 2. Check if the API key is set → yes 3. Check if the endpoint is correct → yes 4. Compare the key character by character →

```
# Agent's configured key:
openclaw-proxy-key-2026

# Proxy's expected key:
nclaw-proxy-key-2026
```

The `o` and `pe` at the start. That's it. Two hours of debugging.

Prevention:

```
#!/bin/bash
# validate-api-keys.sh
# Run this after any configuration change touching API keys

PROXY_KEY=$(grep -oP '"api_key":\s*"\K[^"]+' /opt/cliproxyapi/auth-profiles.json 2>/dev/null)
CLIENT_KEY=$(grep -oP 'ANTHROPIC_API_KEY=\K\S+' ~/.bashrc 2>/dev/null)

if [[ -z "$PROXY_KEY" ]]; then
    echo "ERROR: Could not read proxy API key"
    exit 1
fi

if [[ -z "$CLIENT_KEY" ]]; then
    echo "ERROR: Could not read client API key"
    exit 1
fi

if [[ "$PROXY_KEY" == "$CLIENT_KEY" ]]; then
    echo "OK: API keys match ($PROXY_KEY)"
else
    echo "MISMATCH:"
    echo "  Proxy key:  $PROXY_KEY"
    echo "  Client key: $CLIENT_KEY"
    exit 1
fi
```

More broadly, version control your configuration files. When you can run `git diff` on a config change, typos become visible immediately.

```
# Track agent configs in git
cd ~/.openclaw
git init
git add openclaw.json auth-profiles.json
git commit -m "Initial agent config"

# Now any change shows a diff
```

```
git diff openclaw.json
# Shows: - "nclaw-proxy-key-2026"
#          + "openclaw-proxy-key-2026"
```

Tool Use Patterns for Production

A few patterns that come up repeatedly in production agent systems:

Pattern: Read-Before-Write

Always read a configuration before modifying it. This prevents the agent from creating an invalid file if it doesn't understand the existing format.

```
{
    "name": "update_config",
    "description": (
        "Update a configuration file safely. "
        "Always reads the current content first, "
        "makes minimal targeted changes, "
        "validates the result before writing."
    )
}
```

In the executor:

```
async def update_config(path: str, changes: dict) -> dict:
    # Read current state
    current = read_file(path)
    if not current["success"]:
        return {"success": False, "error": f"Cannot read current config: {current['error']}"}

    # Parse and apply changes
    try:
        config = json.loads(current["content"])
        config.update(changes)
        new_content = json.dumps(config, indent=2)
    except json.JSONDecodeError as e:
        return {"success": False, "error": f"Invalid JSON in current config: {e}"}

    # Validate before writing
    try:
        json.loads(new_content)  # Ensure result is valid JSON
    except json.JSONDecodeError as e:
        return {"success": False, "error": f"Generated invalid JSON: {e}"}

    # Write with backup
    backup_path = f"{path}.bak.{int(time.time())}"
```

```
    write_file(backup_path, current["content"])
    result = write_file(path, new_content)
    result["backup"] = backup_path
    return result
```

Pattern: Dry-Run First

For destructive operations, implement a `dry_run` parameter that shows what would happen without doing it.

```
{
    "name": "cleanup_old_logs",
    "description": "Remove log files older than N days. Use dry_run=true first to preview.",
    "input_schema": {
        "properties": {
            "directory": {"type": "string"},
            "older_than_days": {"type": "integer"},
            "dry_run": {
                "type": "boolean",
                "description": "If true, show what would be deleted without deleting. Default t
                "default": True  # Default to safe behavior
            }
        }
    }
}
```

Pattern: Idempotent Tools

Design tools to be safe to call multiple times with the same input. The model might call a tool again if it's uncertain whether the first call succeeded.

```
def ensure_service_running(service_name: str) -> dict:
    """Start service if not running. Safe to call multiple times."""
    result = subprocess.run(["systemctl", "is-active", service_name],
                            capture_output=True, text=True)

    if result.returncode == 0:
        return {"success": True, "action": "already_running", "service": service_name}

    # Not running - start it
    start_result = subprocess.run(["systemctl", "start", service_name],
                                  capture_output=True, text=True)

    return {
        "success": start_result.returncode == 0,
        "action": "started",
        "service": service_name,
        "error": start_result.stderr if start_result.returncode != 0 else None
    }
```

Key Takeaways

1. **The model decides what to call; your executor decides what it can reach.** All security decisions happen in your code, not the model's reasoning.
2. **Tool descriptions are model instructions.** Write them as explicit decision criteria, including "do NOT use for" clauses.
3. **Three-layer defense for prohibited operations:** system prompt + tool description + executor enforcement. You need all three.
4. **The agent cut its own feet off because nobody told it what was off-limits.** Make the off-limits list explicit, specific, and enforced in code.
5. **Parallel tool execution** is a free performance win for independent operations. Implement it.
6. **Interleaved thinking** (Claude Opus 4.6) makes diagnostic chains auditable and dramatically improves complex multi-step reasoning. Use it for incidents, not status checks.
7. **MCP standardizes tool integration.** If you're building agent infrastructure, invest in MCP servers early—they compose better than custom integrations.
8. **Version control your API key configuration.** Typos happen. `git diff` catches them.
9. **Idempotent tools reduce risk.** Design tools so calling them twice doesn't cause double trouble.
10. **Read before you write.** The model can't know the current state of a file it hasn't read. Don't let it guess.

Next up: Part 3 — Operations. We move from architecture to running agents in production: monitoring, cost control, incident response, and the things that break at 3am.

PART 3: OPERATIONS

Chapter 7: Deploying AI Agents to Production

> "It works on my machine" is a classic developer excuse. With AI agents, the equivalent is "it works when I'm watching." The moment you walk away, something breaks in a way you never anticipated.

Running AI agents in production is fundamentally different from running traditional web services. Agents make decisions, consume API quotas, spawn sub-processes, write files, call external services, and generally behave in ways that are difficult to predict. The infrastructure that supports them needs to be robust, observable, and — critically — safe against the agents themselves doing something unexpected.

This chapter covers how I built and operate a production AI agent infrastructure across three servers. It includes what I got right, what I got catastrophically wrong, and the specific configs and patterns that now keep things running.

The Server Topology

Before getting into deployment patterns, it helps to understand the physical layout. Most of what I'll describe assumes this kind of multi-server setup, though the patterns apply to single-server deployments too.

```
Internet


  Load Balancer (LB)  142.91.102.182
  - Traefik reverse proxy
  - HAProxy (DB connection pooling)
  - Static file server (files.hieudc.com)
  - Claw Đệ (secondary AI agent)

              WireGuard VPN (10.10.0.0/24)
```

```
Server-1     Server-2      Server-3 (VN)
SG           SG            103.199.19.221
(Primary)    (Cognee)      - Monitoring stack
                           - Prometheus/Grafana
OpenClaw     Cognee        - Loki/Tempo
Gateway      API+UI        - Firecrawl
n8n          Neo4j         - yt-dlp
Chatwoot     Postgres
CLIProxy
```

```
Oracle ARM
213.35.108.172
- Cognee MCP server
- Postgres
- Neo4j
```

WireGuard mesh: Full mesh VPN across all 5 nodes. All inter-server traffic routes through encrypted tunnels (10.10.0.0/24). This is what makes VPN-binding for sensitive ports practical.

```
WireGuard IP assignments:
  Server-3 (VN):      10.10.0.1
  Server-1 (SG):      10.10.0.2
  Server-2 (SG):      10.10.0.3
  Load Balancer:       10.10.0.4
  Oracle ARM:          10.10.0.5
```

Latency from Server-1: - Server-2: 0.9ms (same DC, P2P) - Oracle: 2.0ms - Load Balancer: 3.5ms - Server-3 (VN): 54ms (cross-country)

Process Manager Reality Check: Docker vs Systemd vs PM2

When I started running AI agent services, I made the mistake of picking one process manager and trying to use it for everything. Reality forced a more pragmatic approach.

The Service Inventory (Server-1 Reality Check)

Server-1 currently runs: - 14 Docker services - 8 Systemd services - 1 PM2 service (legacy, soon to be migrated)

That's 23 services managed by 3 different process managers on a single server. Is that ideal? No. Is it reality? Yes. Here's the decision logic behind each choice.

When to Use Docker

Docker is the right choice when: - The service has multiple dependencies (DB, Redis, message queue) - You need network isolation between components - The service was designed as a container (official image exists) - You need reproducible environments across servers - Secrets management via environment files is important

```
# docker-compose.yml pattern for agent services
version: "3.8"

services:
  openclaw-gateway:
    image: openclaw/gateway:latest
    container_name: openclaw-gateway
    restart: unless-stopped
    env_file:
      - .env
    environment:
      - NODE_ENV=production
    volumes:
      - ./workspace:/workspace
      - ./logs:/logs
    networks:
      - agent-net
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.openclaw.rule=Host(`api.openclaw.ai`)"
      - "traefik.http.routers.openclaw.tls.certresolver=letsencrypt"
      - "traefik.http.services.openclaw.loadbalancer.server.port=3000"

  redis:
    image: redis:7-alpine
    container_name: agent-redis
    restart: unless-stopped
    # CRITICAL: bind only to VPN interface for security
    command: redis-server --bind 10.10.0.2 127.0.0.1 --requirepass ${REDIS_PASSWORD}
    volumes:
      - redis-data:/data
    networks:
      - agent-net

networks:
  agent-net:
    driver: bridge

volumes:
  redis-data:
```

When to Use Systemd

Systemd is the right choice when: - The service is a single binary or script - It needs access to host resources (hardware, specific kernel features) - Startup order dependencies on host services matter - You need fine-grained resource limits via cgroups - The service predates Docker or containerizing it adds unnecessary complexity

```
# /etc/systemd/system/openclaw.service
[Unit]
Description=OpenClaw AI Agent Gateway
After=network.target
Wants=network-online.target

[Service]
Type=simple
User=hieudc
Group=hieudc
WorkingDirectory=/home/hieudc/.openclaw
ExecStart=/usr/bin/node /home/hieudc/.openclaw/gateway/index.js
ExecReload=/bin/kill -HUP $MAINPID
Restart=on-failure
RestartSec=10
# Prevent runaway resource consumption
MemoryMax=2G
CPUQuota=80%
# Environment
EnvironmentFile=/home/hieudc/.openclaw/.env
# Logging
StandardOutput=journal
StandardError=journal
SyslogIdentifier=openclaw

[Install]
WantedBy=multi-user.target
```

Key systemd features for agent services: - `MemoryMax=2G` — agents can consume unbounded memory if there's a bug in context management - `CPUQuota=80%` — prevents a runaway agent from starving other services - `Restart=on-failure` with `RestartSec=10` — automatic recovery with backoff

When to Use PM2 (And When Not To)

PM2 is appropriate for Node.js services where you want zero-downtime reloads and built-in cluster mode. I still have one service on PM2 (GitPocket API), largely for historical reasons.

For AI agents specifically, PM2 has a weakness: its process monitoring doesn't integrate well with Docker-based monitoring stacks. If you're using Prometheus + Grafana (which you should be), systemd or Docker gives you better observability out of the box.

```
# PM2 config for reference (ecosystem.config.js)
module.exports = {
```

```
  apps: [{
    name: 'gitpocket-api',
    script: './dist/index.js',
    instances: 1,
    autorestart: true,
    watch: false,
    max_memory_restart: '512M',
    env_production: {
      NODE_ENV: 'production',
      PORT: 3001
    }
  }]
}
```

My recommendation: Don't introduce PM2 for new agent services. If you're already using it, migrate gradually to systemd or Docker when you get the chance.

Traefik with docker-socket-proxy (The Right Way)

A common pattern for running Traefik in Docker is to mount the Docker socket directly:

```
# DO NOT DO THIS
volumes:
  - /var/run/docker.sock:/var/run/docker.sock:ro
```

The problem is that anyone who can access the Traefik container effectively has root on the host. Since AI agents can potentially influence container configurations (they have access to the filesystem), this is a real threat vector.

The solution is `docker-socket-proxy`, which sits between Traefik and the Docker socket, exposing only the APIs Traefik actually needs.

```
# docker-compose.yml for Traefik with socket proxy
version: "3.8"

services:
  socket-proxy:
    image: tecnativa/docker-socket-proxy:latest
    container_name: socket-proxy
    restart: unless-stopped
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
    environment:
      # Only expose what Traefik needs
      CONTAINERS: 1
      SERVICES: 1
      NETWORKS: 1
      TASKS: 1
```

```
      # Explicitly deny dangerous APIs
      POST: 0
      AUTH: 0
      SECRETS: 0
      EXEC: 0
    networks:
      - proxy-internal

  traefik:
    image: traefik:v3.0
    container_name: traefik
    restart: unless-stopped
    command:
      - "--api.insecure=false"
      - "--providers.docker=true"
      - "--providers.docker.endpoint=tcp://socket-proxy:2375"
      - "--providers.docker.exposedbydefault=false"
      - "--providers.file.directory=/etc/traefik/dynamic"
      - "--providers.file.watch=true"
      - "--entrypoints.web.address=:80"
      - "--entrypoints.websecure.address=:443"
      - "--certificatesresolvers.letsencrypt.acme.email=admin@docaohieu.com"
      - "--certificatesresolvers.letsencrypt.acme.storage=/letsencrypt/acme.json"
      - "--certificatesresolvers.letsencrypt.acme.tlschallenge=true"
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - letsencrypt:/letsencrypt
      - ./dynamic:/etc/traefik/dynamic:ro
    networks:
      - proxy-internal
      - proxy-public
    depends_on:
      - socket-proxy

networks:
  proxy-internal:
    internal: true  # socket-proxy has no external access
  proxy-public:
    external: true

volumes:
  letsencrypt:
```

Traefik dynamic configuration for services:

```yaml
# /etc/traefik/dynamic/middlewares.yml
http:
  middlewares:
    secure-headers:
      headers:
        stsSeconds: 31536000
        stsIncludeSubdomains: true
        contentTypeNosniff: true
        browserXssFilter: true
        referrerPolicy: "strict-origin-when-cross-origin"

    rate-limit:
      rateLimit:
        average: 100
        burst: 200
        period: 1s

    auth-forward:
      forwardAuth:
        address: "http://auth-service:3000/verify"
        trustForwardHeader: true
```

VPN Binding for Sensitive Ports

One of the most important security practices for a multi-server setup is binding sensitive services to VPN interfaces rather than public IPs. This means the service is only reachable from within the WireGuard network.

Services that should never be publicly accessible: - PostgreSQL (5432) - Redis (6379) - Prometheus (9090) - Grafana (3000, if not public-facing) - Internal agent APIs - Monitoring exporters

```
# PostgreSQL: bind only to VPN and localhost
# /etc/postgresql/14/main/postgresql.conf
listen_addresses = '127.0.0.1,10.10.0.2'

# Redis: similar approach
# /etc/redis/redis.conf
bind 127.0.0.1 10.10.0.2

# Prometheus: VPN-only access
# In systemd unit or docker command
--web.listen-address="10.10.0.2:9090"
```

For Docker services, use explicit port binding:

```yaml
services:
  postgres:
```

```yaml
    image: postgres:16
    ports:
      # Bind only to VPN IP, not 0.0.0.0
      - "10.10.0.2:5432:5432"
    environment:
      POSTGRES_PASSWORD: ${DB_PASSWORD}
```

UFW rules to match:

```
# Allow DB access only from VPN subnet
ufw allow in on wg0 to any port 5432
ufw allow in on wg0 to any port 6379
ufw allow in on wg0 to any port 9090

# Public ports
ufw allow 80/tcp
ufw allow 443/tcp
ufw allow 51820/udp  # WireGuard
```

HAProxy for Database Connection Pooling

AI agents tend to be chatty with databases. A single agent session might open dozens of short-lived connections. Multiply this by several agent instances and you quickly exhaust PostgreSQL's connection limit.

HAProxy solves this with connection pooling at the load balancer level.

```
# /etc/haproxy/haproxy.cfg

global
    log /dev/log local0
    maxconn 4096
    user haproxy
    group haproxy
    daemon

defaults
    mode tcp
    log global
    option tcplog
    option dontlognull
    timeout connect 5s
    timeout client 30s
    timeout server 30s

frontend postgres_frontend
    bind 10.10.0.4:5432
```

```
    default_backend postgres_backend

backend postgres_backend
    balance roundrobin
    option tcp-check
    server db1 10.10.0.2:5432 check inter 2s rise 2 fall 3 maxconn 100
    # Failover to Server-2 if primary is down
    server db2 10.10.0.3:5432 check inter 2s rise 2 fall 3 maxconn 50 backup

frontend redis_frontend
    bind 10.10.0.4:6379
    default_backend redis_backend

backend redis_backend
    balance first
    server redis1 10.10.0.2:6379 check inter 2s maxconn 200
```

With this in place, agent services connect to the HAProxy frontend (10.10.0.4:5432), and HAProxy manages actual PostgreSQL connections efficiently.

War Story #1: The Firecrawl Network Isolation Bug

Date: Early February 2026 **Severity:** Medium — service inaccessible from remote servers **Resolution time:** ~30 minutes once diagnosed

What Happened

I deployed Firecrawl on Server-3 using the official Docker Compose. Everything worked fine when I tested from Server-3 itself. I set up the Traefik routing, the DNS pointed to `firecrawl.docaohieu.com`, and I called it done.

Two days later, I tried to call the Firecrawl API from Server-1 to crawl a website for a knowledge base project. Nothing. Timeout after timeout.

```
# From Server-1:
curl https://firecrawl.docaohieu.com/v1/scrape
# Works (goes through public internet → Traefik → Firecrawl)

# From Server-1 via internal API call:
curl http://10.10.0.1:3002/v1/scrape
# Connection refused
```

The public endpoint worked because it went through the full internet routing. The internal call failed because Firecrawl's API service was listening only on localhost.

Root Cause

The Firecrawl docker-compose.yml had this in the API service configuration:

```yaml
# Default Firecrawl config (the problem)
services:
  api:
    ports:
      - "127.0.0.1:3002:3002"
```

The `127.0.0.1:3002:3002` binding means: expose port 3002 on the host, but only listen on the loopback interface. Any connection from an external IP (including VPN tunnel traffic) gets refused immediately.

The Fix

```yaml
# Fixed configuration
services:
  api:
    ports:
      # 0.0.0.0 means "all interfaces" including VPN
      - "0.0.0.0:3002:3002"
    # Or shorter form:
    # - "3002:3002"
```

Then update UFW to allow access on the VPN interface:

```
# Allow Firecrawl API from VPN network only
ufw allow in on wg0 to any port 3002
# Block from public internet (Traefik handles public access)
ufw deny 3002/tcp
```

Lesson Learned

When a service needs to be callable from other servers in your network, the Docker port binding `127.0.0.1:PORT:PORT` will silently block you. The symptom — connection refused rather than a network error — can mislead you into thinking the service is down when it's actually just unreachable.

Check your port bindings with:

```
docker ps --format "table {{.Names}}\t{{.Ports}}"
# or more detailed:
ss -tlnp | grep docker
```

The lesson extends beyond Firecrawl. Any service you deploy that needs cross-server accessibility should use `0.0.0.0` binding (secured at the firewall level with UFW rules) rather than `127.0.0.1` binding.

War Story #2: The Server That Had No Firewall

Date: February 2026, during a security hardening audit **Severity:** High — server was publicly accessible with no firewall for unknown duration **Resolution time:** ~2 hours (careful hardening to avoid locking ourselves out)

What Happened

I was doing a routine security audit across all three servers. The checklist included verifying that UFW was active and properly configured on each server. Server-1 and Server-2 checked out fine. Then I got to Server-3:

```
# On Server-3:
sudo ufw status
# Status: inactive
```

Not "active with some rules missing." Not "enabled with a permissive policy." Just: **inactive**. The firewall had never been turned on. Server-3, which hosts the monitoring stack (Prometheus, Grafana, Loki, Tempo), had been running with all ports wide open to the internet for its entire operational lifetime.

The Damage Assessment

```
# Check what was actually accessible
ss -tlnp

# Findings:
# 0.0.0.0:9090  → Prometheus (anyone could query your metrics)
# 0.0.0.0:3000  → Grafana (anyone could read your dashboards)
# 0.0.0.0:3100  → Loki (anyone could query your logs)
# 0.0.0.0:9411  → Tempo traces
# 0.0.0.0:9100  → node_exporter (full system metrics)
```

Prometheus metrics are not "dangerous" in the traditional sense — they don't give you direct system access. But they expose a lot of operational intelligence: what services you're running, their health, resource consumption patterns, even internal endpoint names if you're scraping service discovery.

Grafana without authentication is a more serious issue. Anyone who discovered the endpoint could read your dashboards, and if admin credentials were default or weak, they could modify them.

The Safe Hardening Procedure

The dangerous part of enabling UFW on a live server is accidentally blocking SSH and locking yourself out. I've seen this happen. It's not fun.

```
# CRITICAL: Do this in the right order

# Step 1: Allow SSH BEFORE enabling UFW
sudo ufw allow OpenSSH
sudo ufw allow 22/tcp  # belt and suspenders
```

```
# Step 2: Allow WireGuard
sudo ufw allow 51820/udp

# Step 3: Allow public services
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp

# Step 4: Allow monitoring from VPN only
sudo ufw allow in on wg0 to any port 9090   # Prometheus
sudo ufw allow in on wg0 to any port 3000   # Grafana
sudo ufw allow in on wg0 to any port 3100   # Loki

# Step 5: Deny everything else (implicit in UFW, but explicit is better)
sudo ufw default deny incoming
sudo ufw default allow outgoing

# Step 6: Enable - NOW (not before)
sudo ufw enable

# Step 7: Verify immediately
sudo ufw status verbose
```

If you're doing this remotely over SSH, there's still a risk. Test your SSH rule with a new connection (in a separate terminal) before closing your existing session.

```
# In terminal 2, before closing terminal 1:
ssh user@server-3 "echo 'SSH still works'"
```

Only after confirming SSH still works do you close terminal 1.

What We Added to the Audit Checklist

After this incident, every server onboarding includes:

```
#!/bin/bash
# server-hardening-checklist.sh

echo "=== Security Audit for $(hostname) ==="

echo ""
echo "1. UFW Status:"
sudo ufw status | head -5

echo ""
echo "2. Open Ports (all interfaces):"
ss -tlnp | grep LISTEN

echo ""
echo "3. SSH Configuration:"
```

```
grep -E "PasswordAuthentication|PermitRootLogin|PubkeyAuthentication" /etc/ssh/sshd_config

echo ""
echo "4. Docker port bindings:"
docker ps --format "table {{.Names}}\t{{.Ports}}" 2>/dev/null || echo "Docker not running"

echo ""
echo "5. Fail2ban status:"
sudo fail2ban-client status 2>/dev/null || echo "Fail2ban not installed"
```

Run this on every new server before connecting it to your network.

Docker Compose Patterns for Agent Services

Pattern 1: Agent with Isolated Network

```
version: "3.8"

services:
  agent-service:
    image: your-agent:latest
    container_name: agent-service
    restart: unless-stopped
    env_file: .env
    networks:
      - agent-internal
      - proxy-public  # Only if needs external access
    volumes:
      - agent-data:/data
      - /home/hieudc/.openclaw/workspace:/workspace:ro  # Read-only workspace
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.agent.rule=Host(`agent.yourdomain.com`)"
      - "traefik.http.routers.agent.middlewares=rate-limit,secure-headers"
      - "traefik.http.services.agent.loadbalancer.server.port=3000"
    # Resource limits - critical for agents
    deploy:
      resources:
        limits:
          memory: 2G
          cpus: '1.0'

  agent-redis:
    image: redis:7-alpine
    container_name: agent-redis
    restart: unless-stopped
```

```
    command: >
      redis-server
      --requirepass ${REDIS_PASSWORD}
      --maxmemory 512mb
      --maxmemory-policy allkeys-lru
    networks:
      - agent-internal  # Not on proxy-public
    volumes:
      - redis-data:/data

networks:
  agent-internal:
    internal: true
  proxy-public:
    external: true

volumes:
  agent-data:
  redis-data:
```

Pattern 2: Multi-Container Agent Stack

For more complex agent setups with multiple components:

```
version: "3.8"

x-common: &common
  restart: unless-stopped
  logging:
    driver: "json-file"
    options:
      max-size: "100m"
      max-file: "3"

services:
  gateway:
    <<: *common
    image: openclaw/gateway:latest
    depends_on:
      postgres:
        condition: service_healthy
      redis:
        condition: service_started
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
```

```
      start_period: 40s

  worker:
    <<: *common
    image: openclaw/worker:latest
    depends_on:
      - gateway
    scale: 2  # Run 2 worker instances

  postgres:
    <<: *common
    image: postgres:16-alpine
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER}"]
      interval: 10s
      timeout: 5s
      retries: 5
    volumes:
      - postgres-data:/var/lib/postgresql/data
    ports:
      - "10.10.0.2:5432:5432"  # VPN-only

  redis:
    <<: *common
    image: redis:7-alpine
    command: redis-server --requirepass ${REDIS_PASSWORD}
    ports:
      - "10.10.0.2:6379:6379"  # VPN-only

volumes:
  postgres-data:
```

Pattern 3: Systemd Unit for Agent Processes

When running agents as system services (not in Docker):

```
# /etc/systemd/system/agent-worker@.service
# Template unit - create instances with: systemctl start agent-worker@1

[Unit]
Description=AI Agent Worker Instance %i
After=network-online.target
Requires=network-online.target

[Service]
Type=simple
User=hieudc
Group=hieudc
```

```
WorkingDirectory=/home/hieudc/agent

# The @ makes this a template - %i is the instance number
ExecStart=/home/hieudc/.venv/bin/python -m agent.worker --instance %i

# Restart policy
Restart=on-failure
RestartSec=15
StartLimitBurst=3
StartLimitIntervalSec=120

# Resource limits
MemoryMax=1G
CPUQuota=50%
TasksMax=64

# Environment
EnvironmentFile=/home/hieudc/agent/.env
Environment="WORKER_ID=%i"
Environment="LOG_LEVEL=info"

# File descriptor limit (agents can open many connections)
LimitNOFILE=65536

# Logging
StandardOutput=journal
StandardError=journal
SyslogIdentifier=agent-worker-%i

[Install]
WantedBy=multi-user.target
```

Start multiple workers:

```
systemctl enable --now agent-worker@1
systemctl enable --now agent-worker@2
systemctl enable --now agent-worker@3
```

Secrets Management

Never hardcode secrets in docker-compose files or systemd units. Use environment files that are excluded from version control.

```
# .env file (never commit this)
ANTHROPIC_API_KEY=sk-ant-...
OPENAI_API_KEY=sk-...
```

```
REDIS_PASSWORD=...
POSTGRES_PASSWORD=...
TELEGRAM_BOT_TOKEN=...

# .gitignore
.env
*.env
.env.*
!.env.example
```

For a more robust setup, use Vault:

```
# Fetch secrets from Vault at service start
#!/bin/bash
# /usr/local/bin/start-with-vault.sh

export VAULT_ADDR="http://10.10.0.2:8200"
export VAULT_TOKEN=$(cat /etc/vault-token)

# Fetch secrets and export
eval $(vault kv get -format=json secret/agent | \
  jq -r '.data.data | to_entries | .[] | "export \(.key)=\(.value)"')

exec "$@"
```

Then in your systemd unit:

```
ExecStart=/usr/local/bin/start-with-vault.sh /home/hieudc/.venv/bin/python -m agent
```

Health Checks and Dependency Management

AI agent services often have complex startup dependencies. Don't just use `depends_on` with service names — use `condition: service_healthy` with proper health checks.

```
services:
  agent:
    depends_on:
      database:
        condition: service_healthy
      redis:
        condition: service_healthy
      model-api:
        condition: service_started  # External service, can't health check

  database:
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
```

```
      interval: 5s
      timeout: 3s
      retries: 10
      start_period: 30s

  redis:
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
      timeout: 3s
      retries: 5
```

Without this, your agent service might start before the database is ready, fail to connect, and then not retry properly. The container appears "running" but the agent is actually broken.

Deployment Checklist

Before deploying any new agent service to production:

```
Pre-deployment:
- [ ] Port bindings use explicit interface (not 0.0.0.0 for internal services)
- [ ] UFW rules added for any new ports
- [ ] Resource limits set (memory, CPU)
- [ ] .env file present and excluded from git
- [ ] Health check configured
- [ ] Restart policy set to unless-stopped or on-failure

Post-deployment:
- [ ] Verify service is up: docker ps or systemctl status
- [ ] Test from another server (not just localhost)
- [ ] Check logs for startup errors: docker logs or journalctl
- [ ] Verify Traefik routing if public-facing
- [ ] Add to monitoring (see Chapter 9)
- [ ] Document in service inventory

Security:
- [ ] Service runs as non-root user
- [ ] Sensitive ports not exposed to public internet
- [ ] Secrets sourced from env file, not hardcoded
- [ ] Docker socket not mounted unless absolutely necessary
```

Summary

Deploying AI agents to production requires more care than deploying traditional web services. Agents make decisions autonomously, which means infrastructure mistakes that would be caught quickly in traditional deployments can go unnoticed until they cause significant damage.

Key principles from this chapter:

1. **Use the right process manager** — Docker for services with dependencies, systemd for system-level services, and avoid PM2 for new deployments
2. **VPN-bind sensitive ports** — never expose databases or monitoring to the public internet
3. **Use docker-socket-proxy** — don't give Traefik (or any service) direct access to the Docker socket
4. **Check port bindings** — `127.0.0.1:PORT:PORT` vs `0.0.0.0:PORT:PORT` is not obvious but critical for multi-server setups
5. **Enable your firewall** — this sounds obvious until you find a server that's been running without one for months
6. **Set resource limits** — agents can consume unbounded resources; limits prevent one bad agent from taking down your entire server

The next chapter covers cost management, because all this infrastructure needs to run AI models that cost real money.

Chapter 8: Cost Management — Don't Go Bankrupt Running AI Agents

> "I thought it was a small experiment. I just left it running overnight. The next morning I opened Google Cloud Console and felt my stomach drop."

AI infrastructure costs have a unique property that traditional cloud costs don't: they can spike exponentially in hours, not days. A runaway agent that keeps retrying a failed task, or an orphaned process polling an API every few seconds, can rack up hundreds of dollars before anyone notices. Unlike an EC2 instance that charges a predictable hourly rate, LLM API costs are tied to token consumption — and token consumption can be wildly unpredictable.

This chapter is about the real cost of running AI agents in production: the $500 bill I didn't expect, the 91% token reduction I eventually achieved, and the systems I built to ensure it never happens again.

The $500 Gemini Incident

Date: Late 2025 **Bill:** ~$500 USD over approximately 48 hours **Refund:** One-time courtesy refund from Google

What Happened

I was experimenting with a batch processing pipeline — using Gemini Pro to analyze and categorize a large dataset. The job was kicked off manually, ran for a while, and then I got distracted with other work. What I didn't realize was that the job had hit an error partway through and entered a retry loop.

The retry logic was naive: on any error, sleep 5 seconds and try again. The error in this case was a transient API issue that kept resolving and re-triggering. The pipeline kept making API calls, generating output tokens, and billing my account at Gemini Pro rates.

By the time I noticed (checking the Google Cloud Console the next morning), the job had been running for roughly 48 hours and had accumulated just under $500 in charges.

```
# The problematic retry pattern (DON'T do this)
while True:
    try:
        response = gemini_client.generate_content(prompt)
        process(response)
    except Exception as e:
        print(f"Error: {e}, retrying in 5s...")
        time.sleep(5)
        continue  # This just keeps going forever
```

What Should Have Happened

```
import time
import logging
from typing import Optional

def call_with_backoff(
    fn,
    max_retries: int = 5,
    base_delay: float = 1.0,
    max_delay: float = 60.0,
    budget_remaining: Optional[float] = None
) -> any:
    """
    Exponential backoff with budget guard.
    Raises after max_retries or if budget exceeded.
    """
    for attempt in range(max_retries):
        try:
            if budget_remaining is not None and budget_remaining <= 0:
                raise RuntimeError("Budget exhausted, stopping retries")

            result = fn()
            return result

        except RateLimitError:
            delay = min(base_delay * (2 ** attempt), max_delay)
            logging.warning(f"Rate limit hit (attempt {attempt+1}/{max_retries}), "
                            f"backing off {delay:.1f}s")
            time.sleep(delay)

        except BudgetExceededError:
            logging.error("Budget exceeded - stopping immediately")
            raise

        except Exception as e:
            if attempt == max_retries - 1:
```

```
                raise  # Don't swallow the final error
            delay = min(base_delay * (2 ** attempt), max_delay)
            logging.warning(f"Attempt {attempt+1} failed: {e}. Retry in {delay:.1f}s")
            time.sleep(delay)

    raise MaxRetriesExceeded(f"Failed after {max_retries} attempts")
```

Getting the Refund

I contacted Google Cloud Support and explained the situation honestly: a runaway process, I didn't notice it in time, here's the usage graph showing the spike. Google offered a one-time courtesy refund.

The key word is **one-time**. They were explicit about that. If it happens again, you're on your own.

I asked the support agent what they recommend to prevent this. Their answer: budget alerts. Specifically, set them low enough that you get notified while you still have time to react.

Budget Alerts Are Not Optional

Every AI API you use should have budget alerts configured. This is not a nice-to-have. It is the first thing you set up, before you write a single line of code.

Google Cloud Budget Alerts

```
# Create budget via gcloud CLI
gcloud billing budgets create \
  --billing-account=YOUR_BILLING_ACCOUNT_ID \
  --display-name="AI API Monthly Budget" \
  --budget-amount=100USD \
  --threshold-rule=percent=50,basis=CURRENT_SPEND \
  --threshold-rule=percent=80,basis=CURRENT_SPEND \
  --threshold-rule=percent=100,basis=CURRENT_SPEND \
  --all-updates-rule-pubsub-topic=projects/YOUR_PROJECT/topics/budget-alerts
```

But alerts alone aren't enough — you need to act on them. Set up a Pub/Sub subscriber that actually does something:

```
# budget_alert_handler.py
import json
import requests
from google.cloud import pubsub_v1

TELEGRAM_BOT_TOKEN = "your-bot-token"
TELEGRAM_CHAT_ID = "your-chat-id"
ALERT_THRESHOLD_KILL = 90  # Kill expensive processes at 90% budget
```

```
def handle_budget_alert(message):
    data = json.loads(message.data.decode("utf-8"))

    budget_amount = float(data["budgetAmount"])
    cost_amount = float(data["costAmount"])
    percent_used = (cost_amount / budget_amount) * 100

    alert_text = (
        f"BUDGET ALERT\n"
        f"Spent: ${cost_amount:.2f} / ${budget_amount:.2f}\n"
        f"Usage: {percent_used:.1f}%\n"
        f"Period: {data.get('budgetDisplayName', 'unknown')}"
    )

    # Send Telegram notification
    requests.post(
        f"https://api.telegram.org/bot{TELEGRAM_BOT_TOKEN}/sendMessage",
        json={"chat_id": TELEGRAM_CHAT_ID, "text": alert_text}
    )

    # At 90%, kill non-critical batch jobs
    if percent_used >= ALERT_THRESHOLD_KILL:
        kill_batch_jobs()

def kill_batch_jobs():
    """Stop any running batch processing jobs."""
    import subprocess
    # Kill any processes tagged as batch jobs
    subprocess.run(["pkill", "-f", "batch_processor"], check=False)
    # Send additional alert
    requests.post(
        f"https://api.telegram.org/bot{TELEGRAM_BOT_TOKEN}/sendMessage",
        json={
            "chat_id": TELEGRAM_CHAT_ID,
            "text": "AUTO-KILL: Batch jobs terminated to prevent budget overrun"
        }
    )
```

Anthropic Claude Budgets

Anthropic doesn't have the same real-time budget API as Google Cloud, but you can set spending limits in the console and use usage-based rate limiting on your side:

```
# cost_tracker.py - Track Claude API spend in real-time
import anthropic
from datetime import datetime
import json
```

```
import os

DAILY_BUDGET_USD = 10.0
COST_FILE = "/tmp/claude_daily_cost.json"

# Approximate costs per 1M tokens (Claude Sonnet 4.6 as of early 2026)
COSTS = {
    "claude-opus-4": {"input": 15.0, "output": 75.0},
    "claude-sonnet-4": {"input": 3.0, "output": 15.0},
    "claude-haiku-4": {"input": 0.25, "output": 1.25},
}

def load_daily_cost() -> dict:
    today = datetime.now().strftime("%Y-%m-%d")
    if os.path.exists(COST_FILE):
        with open(COST_FILE) as f:
            data = json.load(f)
            if data.get("date") == today:
                return data
    return {"date": today, "total_usd": 0.0, "requests": 0}

def save_daily_cost(data: dict):
    with open(COST_FILE, "w") as f:
        json.dump(data, f)

def estimate_cost(model: str, input_tokens: int, output_tokens: int) -> float:
    pricing = COSTS.get(model, COSTS["claude-sonnet-4"])
    return (
        input_tokens / 1_000_000 * pricing["input"] +
        output_tokens / 1_000_000 * pricing["output"]
    )

def call_claude_with_budget_check(client, model, messages, **kwargs):
    cost_data = load_daily_cost()

    if cost_data["total_usd"] >= DAILY_BUDGET_USD:
        raise RuntimeError(
            f"Daily budget ${DAILY_BUDGET_USD} exceeded. "
            f"Spent: ${cost_data['total_usd']:.4f}"
        )

    response = client.messages.create(
        model=model,
        messages=messages,
        **kwargs
    )
```

```python
    # Track cost
    cost = estimate_cost(
        model,
        response.usage.input_tokens,
        response.usage.output_tokens
    )
    cost_data["total_usd"] += cost
    cost_data["requests"] += 1
    save_daily_cost(cost_data)

    return response
```

Trial Credits and the Models That Don't Count

One painful discovery: **trial credits don't apply to all models.** When I first signed up for Google AI Studio, I had $300 in trial credits. I assumed this covered everything. It didn't.

Certain Gemini models — specifically the ones most useful for complex agent tasks — were billed separately even during the trial period. I discovered this when I got a small bill ($12) during what I thought was my free trial period.

Google AI Pricing (approximate, verify current pricing):

Model	Input (per 1M)	Output (per 1M)
Gemini 2.0 Flash	$0.075	$0.30
Gemini 2.5 Flash	$0.15	$0.60
Gemini 2.5 Pro	$1.25 (<200K)	$10.00
Gemini 3 Flash	$0.15	$0.60
Gemini 3 Pro	$2.50	$15.00

Anthropic Claude Pricing (approximate):

Model	Input (per 1M)	Output (per 1M)
Claude Haiku 4	$0.25	$1.25
Claude Sonnet 4.6	$3.00	$15.00
Claude Opus 4	$15.00	$75.00

Rule: Always test with the cheapest model that works. Only upgrade to more expensive models when you have a specific capability requirement that the cheaper models fail to meet.

The Token Optimization Crisis

In February 2026, fifteen of my Google Antigravity accounts were suspended simultaneously. This cut my free API access from 750 requests/week to essentially zero overnight. It forced an emergency token optimization sprint.

The Problem: Baseline Token Consumption

Before optimization, here's what was being injected into every single agent context:

```
Context files injected per request:

  File             Tokens     Contents

  AGENTS.md        ~3,535     Full agent roster with examples
  MEMORY.md        ~2,833     Full operational memory
  TOOLS.md         ~838       Complete tool documentation
  SOUL.md          ~636       Extended personality guidelines
  HEARTBEAT.md     ~569       Full heartbeat logic
  BOOTSTRAP.md     ~360       Startup procedures
  USER.md          ~158       User preferences
  IDENTITY.md      ~101       Identity configuration

  TOTAL            ~9,030
```

9,030 tokens per request, just for the system context. With dozens of requests per hour, that adds up fast.

The Optimization

The insight was that most of this context was reference information — details you only need when actively working on a specific task, not on every single call.

```
After optimization:

  File             Before     After        Reduction

  AGENTS.md        ~3,535     ~270         -92%
  MEMORY.md        ~2,833     ~163         -94%
  TOOLS.md         ~838       ~105         -87%
  SOUL.md          ~636       ~113         -82%
  HEARTBEAT.md     ~569       ~23          -96%
  BOOTSTRAP.md     ~360       ~39          -89%
  USER.md          ~158       ~47          -70%
  IDENTITY.md      ~101       ~37          -63%

  TOTAL            ~9,030     ~797         -91%
```

91% reduction. From ~9,000 tokens to ~800 tokens per context load.

How? The key principle is: **don't inject reference data, inject pointers to reference data.**

Instead of:

```
# AGENTS.md (3,535 tokens)
## Available Agents

### researcher
The researcher agent conducts deep technical research...
[hundreds of lines of documentation]

### planner
The planner agent creates implementation plans...
[hundreds more lines]

### code-reviewer
...
```

After:

```
# AGENTS.md (270 tokens)
Agents: researcher, planner, code-reviewer, tester, debugger, docs-manager
Full docs: docs/agents/
Use: spawn <agent-name> for specialized tasks
```

The agent can still access full documentation when it needs it — it just doesn't load all of it on every single call.

Heartbeat Interval Optimization

The OpenClaw agent had a heartbeat routine that ran every 3 minutes. Each heartbeat triggered context loads, memory syncs, and health checks. That's 20 API calls per hour just for heartbeats, burning tokens even when the agent was idle.

The fix: increase the interval and separate light heartbeats from heavy operations.

```
Before:
- Heartbeat every 3 minutes
- Each heartbeat: load full context + run health checks + sync memory
- Cost: ~20 heavy API calls/hour during idle

After:
- Heartbeat every 30 minutes
- Each heartbeat: minimal status check only
- Health checks: moved to systemd timers (zero AI tokens)
- Memory sync: triggered on events, not on schedule
- Cost: ~2 light API calls/hour during idle
```

The heartbeat went from burning 20+ API calls/hour to 2. That's a 10x reduction in idle cost.

Deleting All Auto Cron Jobs

The most radical decision was to delete all automatic AI cron jobs. Every single one.

Here's the reasoning: each health check cron job was running the AI model to do things that didn't require intelligence. Checking if a container is up is a solved problem — `docker inspect` can tell you in milliseconds. Sending a Telegram alert is a simple HTTP call. None of that requires an AI model.

```
# Before: OpenClaw cron jobs that ran every few minutes
# These each consume AI tokens:
# - auto-heal-alerts (every 1 min)
# - memory-auto-save (every 5 min)
# - devops-health-check (every 6h)

# After: Systemd timers that cost zero AI tokens
```

```
# /etc/systemd/system/health-check.service
[Unit]
Description=Server Health Check (no AI)
After=network.target

[Service]
Type=oneshot
User=hieudc
ExecStart=/home/hieudc/scripts/health-check.sh
```

```
# /etc/systemd/system/health-check.timer
[Unit]
Description=Run health check every 5 minutes

[Timer]
OnBootSec=60
OnUnitActiveSec=5min
Persistent=true

[Install]
WantedBy=timers.target
```

```
#!/bin/bash
# /home/hieudc/scripts/health-check.sh
# Pure bash health check - zero AI tokens

TELEGRAM_BOT_TOKEN="${TELEGRAM_BOT_TOKEN}"
TELEGRAM_CHAT_ID="${TELEGRAM_CHAT_ID}"

send_alert() {
    local message="$1"
    curl -s -X POST \
        "https://api.telegram.org/bot${TELEGRAM_BOT_TOKEN}/sendMessage" \
```

```
        -d "chat_id=${TELEGRAM_CHAT_ID}&text=${message}" > /dev/null
}

# Check critical containers
CRITICAL_CONTAINERS=("openclaw-gateway" "traefik" "postgres" "redis")

for container in "${CRITICAL_CONTAINERS[@]}"; do
    STATUS=$(docker inspect --format='{{.State.Status}}' "$container" 2>/dev/null)
    if [ "$STATUS" != "running" ]; then
        # Try to restart
        docker start "$container" 2>/dev/null
        NEW_STATUS=$(docker inspect --format='{{.State.Status}}' "$container" 2>/dev/null)
        if [ "$NEW_STATUS" != "running" ]; then
            send_alert "ALERT: ${container} is ${STATUS} and could not be restarted"
        else
            send_alert "INFO: ${container} was ${STATUS}, successfully restarted"
        fi
    fi
done
```

The only time you should use AI in a health check is when you need AI capabilities — diagnosing a complex issue, generating a remediation plan, something that actually requires intelligence. Checking if a port is open doesn't qualify.

Orphan tmux Sessions: The Silent Quota Killer

This one took me a while to diagnose. I kept noticing quota consumption that I couldn't account for. Checking active sessions showed nothing obvious. Then I looked at tmux:

```
tmux ls
# OUTPUT:
# claude-code: 1 windows (created Thu Feb  6 14:23:11 2026)
# debug-session: 1 windows (created Wed Feb  5 09:11:44 2026)
# batch-process: 1 windows (created Mon Feb  3 18:45:22 2026)
# old-experiment: 1 windows (created Sun Feb  2 22:30:01 2026)
```

Four tmux sessions, some days old. Inside each one, a Claude Code or OpenClaw process was still running, waiting for input, periodically refreshing context, consuming tokens on its heartbeat cycle.

The `old-experiment` session from February 2nd had been silently burning quota for four days.

```
# Find and audit all tmux sessions
tmux ls -F "#{session_name}: created #{session_created_string}"

# Look inside each session
tmux capture-pane -p -t claude-code

# Clean up old sessions that shouldn't be running
```

```
tmux kill-session -t old-experiment
tmux kill-session -t debug-session
```

Now I run a weekly audit:

```
#!/bin/bash
# weekly-session-audit.sh

echo "=== Active tmux sessions ==="
tmux ls 2>/dev/null || echo "No tmux sessions"

echo ""
echo "=== Claude Code processes ==="
pgrep -la "claude" 2>/dev/null || echo "No Claude processes"

echo ""
echo "=== OpenClaw processes ==="
pgrep -la "openclaw\|gateway" 2>/dev/null || echo "No OpenClaw processes"

echo ""
echo "=== Orphan Python AI scripts ==="
pgrep -la "python" | grep -E "batch|process|generate|crawl" || echo "None found"
```

CLIProxyAPI: Free Quota Through Account Rotation

After losing my Antigravity accounts, I rebuilt a more sustainable free API strategy using CLIProxyAPI — a proxy that round-robins through multiple Google AI Pro accounts.

How It Works

```
Claude Code / OpenClaw


CLIProxyAPI (localhost:8317)

          Account 1 (quota remaining: 47/50)
          Account 2 (quota remaining: 50/50)
          Account 3 (quota remaining: 0/50) ← exhausted
          Account 4 (quota remaining: 31/50)
          ... (13 accounts total)


Google AI Studio API (free tier)
```

The proxy uses a `fill-first` strategy: exhaust one account completely before switching to the next. This maximizes quota utilization per account since Google's free tier resets weekly.

```bash
# Check which account is currently active
journalctl -u cliproxyapi --no-pager -n 30 | grep "Use OAuth"
# Output: Use OAuth provider=antigravity auth_file=account3@gmail.com.json

# See request distribution across accounts
journalctl -u cliproxyapi --no-pager --since "1 week ago" | \
  grep "Use OAuth" | \
  awk '{print $NF}' | \
  sort | uniq -c | sort -rn
```

The Quota Reality

Google AI Pro quota: 50 requests/week per account.

Not 50 per day. Per week. This surprised me — I expected at least daily resets. The practical math:

```
13 accounts × 50 requests/week = 650 requests/week total
650 / 7 days = ~93 requests/day maximum

With complex agent tasks averaging 5-10 requests each:
= 9-18 meaningful agent tasks per day (free tier)
```

This is enough for development and moderate production use, but a busy AI agent in production will exhaust this quickly. Plan accordingly.

Latency Trade-off

There's a real latency penalty with CLIProxyAPI compared to direct API calls:

```
Latency Comparison:

  Method                        Latency           Cost

  Direct Gemini Flash API        680-914ms         $0.075/1M tokens
  CLIProxyAPI (Antigravity)     2000-2300ms        Free
```

CLIProxyAPI is 2-3x slower because of OAuth overhead and proxy routing. For interactive agent tasks, this is noticeable. For batch processing where you don't care about latency, it's a reasonable trade-off for free access.

Model Tiering Strategy

Not every task needs the most capable (and expensive) model. A tiering strategy routes tasks to the cheapest model that can handle them.

```
Model Tier Decision Matrix:
```

```
TASK TYPE                        MODEL             RATIONALE

Complex reasoning, planning      Claude Opus 4     Quality critical
Code generation (complex)        Claude Sonnet     Balance
Code generation (simple)         Claude Haiku      Speed + cost
Text classification              Gemini Flash      Cheap + fast
Summarization                    Gemini Flash      High volume
Embedding generation             text-embedding    Batch-optimized
Image analysis                   Gemini Pro        Vision required
Simple Q&A / routing             Claude Haiku      Cheapest capable
```

Implementation in code:

```python
from enum import Enum
from typing import Literal

class TaskComplexity(Enum):
    SIMPLE = "simple"       # Classification, routing, yes/no
    MODERATE = "moderate"   # Code generation, summarization
    COMPLEX = "complex"     # Reasoning, planning, analysis
    CRITICAL = "critical"   # Production decisions, security

MODEL_TIERS = {
    TaskComplexity.SIMPLE: "claude-haiku-4",
    TaskComplexity.MODERATE: "claude-sonnet-4-6",
    TaskComplexity.COMPLEX: "claude-opus-4",
    TaskComplexity.CRITICAL: "claude-opus-4",
}

def select_model(complexity: TaskComplexity) -> str:
    """Route to cheapest appropriate model."""
    return MODEL_TIERS[complexity]

# Usage
def analyze_log_entry(log_line: str) -> dict:
    # Log classification is simple - use Haiku
    model = select_model(TaskComplexity.SIMPLE)
    # ...

def generate_deployment_plan(requirements: str) -> str:
    # Deployment planning is complex - use Opus
    model = select_model(TaskComplexity.COMPLEX)
    # ...
```

Batch API: 50% Discount

Both Claude and Gemini offer batch processing APIs with significant discounts. If you have work that doesn't need real-time responses — generating descriptions for 1000 products, analyzing logs for the past week, creating test cases — batch API can cut costs in half.

```
# Anthropic Batch API example
import anthropic

client = anthropic.Anthropic()

# Submit batch of requests
batch = client.messages.batches.create(
    requests=[
        {
            "custom_id": f"request-{i}",
            "params": {
                "model": "claude-haiku-4",
                "max_tokens": 500,
                "messages": [
                    {"role": "user", "content": item}
                ]
            }
        }
        for i, item in enumerate(items_to_process)
    ]
)

print(f"Batch ID: {batch.id}")
print(f"Status: {batch.processing_status}")
# Results available within 24h, at 50% of normal cost
```

```
# Google Gemini Batch API
import google.generativeai as genai

# Create batch prediction job
batch_job = genai.create_batch_prediction_job(
    model="models/gemini-2.0-flash",
    input_config={
        "instances_format": "jsonl",
        "gcs_source": {"uris": ["gs://bucket/input.jsonl"]}
    },
    output_config={
        "predictions_format": "jsonl",
        "gcs_destination": {"output_uri_prefix": "gs://bucket/output/"}
    }
)
# 50% discount vs synchronous API
```

Cost Monitoring Dashboard

Here's a Prometheus + Grafana setup for tracking AI API costs in real-time:

```
# cost_metrics_exporter.py
from prometheus_client import start_http_server, Counter, Gauge, Histogram
import time

# Define metrics
api_tokens_total = Counter(
    'ai_api_tokens_total',
    'Total tokens consumed',
    ['provider', 'model', 'token_type']
)

api_cost_total = Counter(
    'ai_api_cost_usd_total',
    'Total API cost in USD',
    ['provider', 'model']
)

api_request_duration = Histogram(
    'ai_api_request_duration_seconds',
    'API request latency',
    ['provider', 'model'],
    buckets=[0.1, 0.5, 1.0, 2.0, 5.0, 10.0, 30.0]
)

daily_budget_remaining = Gauge(
    'ai_daily_budget_remaining_usd',
    'Remaining daily budget in USD',
    ['provider']
)

def track_api_call(provider: str, model: str, input_tokens: int,
                   output_tokens: int, duration_seconds: float, cost_usd: float):
    """Call this after every API request to track metrics."""
    api_tokens_total.labels(provider, model, 'input').inc(input_tokens)
    api_tokens_total.labels(provider, model, 'output').inc(output_tokens)
    api_cost_total.labels(provider, model).inc(cost_usd)
    api_request_duration.labels(provider, model).observe(duration_seconds)

if __name__ == "__main__":
    start_http_server(8091)
    while True:
        time.sleep(60)
```

Prometheus scrape config:

```
# prometheus.yml (add to scrape_configs)
scrape_configs:
  - job_name: 'ai_cost_metrics'
    static_configs:
      - targets: ['10.10.0.2:8091']
    scrape_interval: 60s
```

Grafana dashboard (key panels):

```
Panel 1: Daily spend by provider (bar chart, last 30 days)
  query: increase(ai_api_cost_usd_total[1d])

Panel 2: Token consumption rate (time series)
  query: rate(ai_api_tokens_total[5m]) * 300

Panel 3: Budget remaining (gauge)
  query: ai_daily_budget_remaining_usd

Panel 4: Cost by model (pie chart)
  query: sum by (model) (ai_api_cost_usd_total)

Panel 5: P95 latency by provider (time series)
  query: histogram_quantile(0.95, rate(ai_api_request_duration_seconds_bucket[5m]))
```

The Cost of "Just Checking"

One pattern I see constantly — including in my own setup before I caught it — is agents that use AI for tasks that don't need AI.

```
# Common anti-pattern: using AI for simple checks
async def is_server_healthy(server_ip: str) -> bool:
    response = await call_claude(
        f"Check if server {server_ip} is healthy and responding"
    )
    # This costs tokens for something ping/curl could do instantly
```

```
# Better: Use AI only when you actually need intelligence
import subprocess
import httpx

def is_server_healthy(server_ip: str, port: int = 80) -> bool:
    """Zero AI cost health check."""
    try:
        result = subprocess.run(
            ["ping", "-c", "1", "-W", "2", server_ip],
            capture_output=True,
```

```
            timeout=3
        )
        return result.returncode == 0
    except subprocess.TimeoutExpired:
        return False

async def diagnose_unhealthy_server(server_ip: str, symptoms: str) -> str:
    """Use AI only when you have a complex diagnosis to make."""
    return await call_claude(
        f"Server {server_ip} is down with symptoms: {symptoms}. "
        f"What are the most likely causes and remediation steps?"
    )
```

The rule: if you can do it with bash, do it with bash.

Budget Monitoring Script

A practical script for daily cost review:

```
#!/bin/bash
# /home/hieudc/scripts/daily-cost-report.sh
# Run via cron: 0 8 * * * /home/hieudc/scripts/daily-cost-report.sh

TELEGRAM_BOT_TOKEN="${TELEGRAM_BOT_TOKEN}"
TELEGRAM_CHAT_ID="${TELEGRAM_CHAT_ID}"

# Collect metrics from Prometheus
PROMETHEUS_URL="http://10.10.0.2:9090"

query_prometheus() {
    local query="$1"
    curl -s "${PROMETHEUS_URL}/api/v1/query" \
        --data-urlencode "query=${query}" | \
        python3 -c "import sys,json; d=json.load(sys.stdin); print(d['data']['result'][0]['valu
}

DAILY_COST=$(query_prometheus "increase(ai_api_cost_usd_total[24h])")
DAILY_TOKENS=$(query_prometheus "increase(ai_api_tokens_total[24h])")
REQUESTS_TODAY=$(query_prometheus "increase(ai_api_requests_total[24h])")

# Get CLIProxyAPI account status
ACTIVE_ACCOUNT=$(journalctl -u cliproxyapi --no-pager --since "1h ago" | \
    grep "Use OAuth" | tail -1 | awk '{print $NF}' | sed 's/.json//')

MESSAGE="Daily AI Cost Report
Date: $(date +%Y-%m-%d)
```

```
Cost: \$${DAILY_COST:-0.00}
Tokens: ${DAILY_TOKENS:-0}
Requests: ${REQUESTS_TODAY:-0}
CLIProxy Active: ${ACTIVE_ACCOUNT:-unknown}"

curl -s -X POST \
    "https://api.telegram.org/bot${TELEGRAM_BOT_TOKEN}/sendMessage" \
    -d "chat_id=${TELEGRAM_CHAT_ID}&text=${MESSAGE}" > /dev/null

echo "Report sent: $MESSAGE"
```

Key Cost Management Rules

After running AI agents in production for months and paying for mistakes like the $500 Gemini incident, here are the rules I follow without exception:

1. **Set budget alerts on every API account, immediately, before writing code**
2. **Use exponential backoff with maximum retry limits** — never retry indefinitely
3. **Route tasks to the cheapest model that can handle them** — use Haiku for simple work, Opus only when necessary
4. **Run infrastructure health checks with bash, not AI** — every AI health check is a token burn
5. **Audit tmux sessions weekly** — orphan sessions silently consume quota
6. **Use batch APIs for non-real-time work** — 50% discount is significant at scale
7. **Monitor token consumption in Prometheus** — you can't manage what you don't measure
8. **Read the fine print on trial credits** — they often don't apply to premium models
9. **Keep context files lean** — inject pointers, not full documentation
10. **Heartbeat intervals should be long** — 30 minutes is fine for most agent heartbeats; 3 minutes is wasteful

The $500 Gemini bill was an expensive lesson. The 91% context reduction that followed saved far more than that over subsequent months. Cost management isn't just about not overspending — it's about building systems that are sustainable to operate at scale.

Chapter 9: Monitoring AI Agents in Production

> "Traditional monitoring asks: is the service up? AI agent monitoring asks: is the service up, is it doing the right thing, is it doing it efficiently, and is it about to bankrupt you?"

Monitoring AI agents is harder than monitoring traditional services. A web server is either responding or it isn't. An AI agent can be technically running — processes healthy, memory fine, CPU normal — while simultaneously doing something completely wrong: stuck in a loop, consuming tokens at 10x normal rate, silently failing on every task, or running as two duplicate instances causing cascading restarts.

This chapter covers the monitoring stack I built, the specific failure modes you need to watch for, and the war stories that taught me what to instrument.

What You Actually Need to Monitor

Before talking about tools, it's worth being specific about what "monitoring AI agents" means. There are five categories:

```
AI Agent Observability Stack


  1. HEALTH         Is the agent process alive and responsive?
     - Process status
     - HTTP health endpoint
     - Memory/CPU consumption

  2. TOKENS         What is the agent consuming?
     - Input/output tokens per request
     - Daily/weekly totals by model
     - Anomaly detection (sudden spikes)

  3. ERRORS         What is failing?
     - API error rates (4xx, 5xx by provider)
     - Rate limit hits (429s)
     - Context limit errors (200K exceeded)
```

```
      - Tool call failures

   4. LATENCY        How fast is the agent responding?
      - Time to first token
      - Total task completion time
      - Queue depth (if using task queues)

   5. COST           What is this costing?
      - Real-time USD spend
      - Cost per task type
      - Budget remaining
```

Most teams start with just health monitoring and add the rest only after something goes wrong. The smarter approach is to instrument all five from day one — the overhead is minimal, and the time saved on debugging will repay it within weeks.

The Monitoring Stack

My stack on Server-3 (VN):

```
Prometheus → scrapes metrics from all servers
Grafana    → dashboards and alerting
Loki       → log aggregation
Tempo      → distributed traces
Alertmanager → routes alerts to Telegram
```

All bound to VPN interfaces (10.10.0.0/24), not publicly accessible.

Prometheus Configuration

```yaml
# /etc/prometheus/prometheus.yml

global:
  scrape_interval: 15s
  evaluation_interval: 15s
  external_labels:
    cluster: 'production'
    region: 'sg-vn'

alerting:
  alertmanagers:
    - static_configs:
        - targets: ['10.10.0.1:9093']

rule_files:
  - "rules/agent-health.yml"
```

```
  - "rules/cost-alerts.yml"
  - "rules/container-alerts.yml"

scrape_configs:
  # Server-1 (SG - Primary)
  - job_name: 'server-1-node'
    static_configs:
      - targets: ['10.10.0.2:9100']
    relabel_configs:
      - target_label: server
        replacement: server-1

  - job_name: 'server-1-docker'
    static_configs:
      - targets: ['10.10.0.2:9323']  # cadvisor
    relabel_configs:
      - target_label: server
        replacement: server-1

  - job_name: 'server-1-agent'
    static_configs:
      - targets: ['10.10.0.2:8091']  # AI cost exporter
    relabel_configs:
      - target_label: server
        replacement: server-1

  # Server-2 (SG - Cognee)
  - job_name: 'server-2-node'
    static_configs:
      - targets: ['10.10.0.3:9100']
    relabel_configs:
      - target_label: server
        replacement: server-2

  # Server-3 (VN - Monitoring)
  - job_name: 'server-3-node'
    static_configs:
      - targets: ['10.10.0.1:9100']
    relabel_configs:
      - target_label: server
        replacement: server-3

  # Load Balancer
  - job_name: 'lb-node'
    static_configs:
      - targets: ['10.10.0.4:9100']
    relabel_configs:
```

```
      - target_label: server
        replacement: loadbalancer

  # Oracle ARM
  - job_name: 'oracle-node'
    static_configs:
      - targets: ['10.10.0.5:9100']
    relabel_configs:
      - target_label: server
        replacement: oracle
```

Alert Rules for Agent Services

```
# /etc/prometheus/rules/agent-health.yml

groups:
  - name: agent_health
    interval: 30s
    rules:
      # Container down alert
      - alert: ContainerDown
        expr: |
          absent(container_last_seen{name=~"openclaw.*|traefik|postgres|redis"})
          or
          time() - container_last_seen{name=~"openclaw.*|traefik|postgres|redis"} > 120
        for: 2m
        labels:
          severity: critical
        annotations:
          summary: "Container {{ $labels.name }} is down"
          description: "Container {{ $labels.name }} on {{ $labels.instance }} has been down fo

      # High restart count - indicates crash loop
      - alert: ContainerRestartingFrequently
        expr: |
          increase(container_restart_count{name!=""}[1h]) > 5
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Container {{ $labels.name }} restarting frequently"
          description: "Container {{ $labels.name }} has restarted {{ $value }} times in the la

      # Memory usage too high
      - alert: AgentHighMemory
        expr: |
          container_memory_usage_bytes{name=~"openclaw.*|agent.*"}
```

```
        / container_spec_memory_limit_bytes{name=~"openclaw.*|agent.*"} > 0.85
      for: 5m
      labels:
        severity: warning
      annotations:
        summary: "Agent {{ $labels.name }} memory usage high"
        description: "Agent {{ $labels.name }} is using {{ $value | humanizePercentage }} of

    # CPU spike - possible runaway loop
    - alert: AgentCPUSpike
      expr: |
        rate(container_cpu_usage_seconds_total{name=~"openclaw.*|agent.*"}[5m]) * 100 > 80
      for: 10m
      labels:
        severity: warning
      annotations:
        summary: "Agent {{ $labels.name }} CPU spike"
        description: "Agent {{ $labels.name }} has been at {{ $value }}% CPU for 10+ minutes

- name: cost_alerts
  rules:
    # Daily spend threshold
    - alert: HighDailyAICost
      expr: |
        increase(ai_api_cost_usd_total[24h]) > 5
      labels:
        severity: warning
      annotations:
        summary: "High AI API daily spend"
        description: "Daily AI API spend is ${{ $value | humanize }}. Check for runaway proc

    # Token spike - 5x normal rate
    - alert: TokenConsumptionSpike
      expr: |
        rate(ai_api_tokens_total[5m]) >
        avg_over_time(rate(ai_api_tokens_total[5m])[1h:5m]) * 5
      for: 5m
      labels:
        severity: critical
      annotations:
        summary: "AI token consumption spike detected"
        description: "Token consumption is 5x the 1-hour average. Possible runaway process."
```

Alertmanager Configuration

```
# /etc/alertmanager/alertmanager.yml
```

```
global:
  resolve_timeout: 5m

route:
  group_by: ['alertname', 'server']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 4h
  receiver: 'telegram-critical'

  routes:
    - match:
        severity: critical
      receiver: 'telegram-critical'
      repeat_interval: 1h

    - match:
        severity: warning
      receiver: 'telegram-warning'
      repeat_interval: 6h

receivers:
  - name: 'telegram-critical'
    telegram_configs:
      - bot_token: '${TELEGRAM_BOT_TOKEN}'
        chat_id: ${TELEGRAM_CHAT_ID}
        message: |
            *CRITICAL ALERT*
          *Alert:* {{ .GroupLabels.alertname }}
          *Server:* {{ .CommonLabels.server }}
          {{ range .Alerts }}
          *Summary:* {{ .Annotations.summary }}
          *Description:* {{ .Annotations.description }}
          {{ end }}
        parse_mode: 'Markdown'

  - name: 'telegram-warning'
    telegram_configs:
      - bot_token: '${TELEGRAM_BOT_TOKEN}'
        chat_id: ${TELEGRAM_CHAT_ID}
        message: |
            *WARNING*
          *Alert:* {{ .GroupLabels.alertname }}
          {{ range .Alerts }}
          {{ .Annotations.summary }}
          {{ end }}
        parse_mode: 'Markdown'
```

```
inhibit_rules:
  # Don't alert on individual service issues if the whole server is down
  - source_match:
      severity: 'critical'
      alertname: 'NodeDown'
    target_match:
      severity: 'warning'
    equal: ['server']
```

Docker Events Listener: Real-Time Container Alerts

Prometheus scrapes on a 15-second interval. That means a container can be down for up to 15 seconds before Prometheus even knows about it, and then the alert needs to fire and route through Alertmanager before you're notified. In practice, notification latency can be 2-5 minutes.

For critical agent services, that's too slow. I built a Docker events listener that fires Telegram alerts in under 5 seconds.

```
#!/usr/bin/env python3
# /home/hieudc/scripts/docker-events-listener.py
# Systemd service: docker-events-listener.service

import docker
import requests
import json
import time
import logging
from datetime import datetime

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

TELEGRAM_BOT_TOKEN = "your-bot-token"
TELEGRAM_CHAT_ID = "your-chat-id"

# Only alert on these containers (others are noisy/expected to restart)
WATCHED_CONTAINERS = {
    "openclaw-gateway",
    "traefik",
    "postgres",
    "redis",
    "n8n",
    "chatwoot",
    "prometheus",
```

```
    "grafana",
}

# Events that warrant alerts
ALERT_EVENTS = {"die", "kill", "oom", "stop"}
INFO_EVENTS = {"start", "restart"}

def send_telegram(message: str, critical: bool = False):
    """Send Telegram notification."""
    prefix = "🚨" if critical else "ℹ️"
    full_message = f"{prefix} *Docker Event*\n{message}"

    try:
        response = requests.post(
            f"https://api.telegram.org/bot{TELEGRAM_BOT_TOKEN}/sendMessage",
            json={
                "chat_id": TELEGRAM_CHAT_ID,
                "text": full_message,
                "parse_mode": "Markdown"
            },
            timeout=10
        )
        response.raise_for_status()
    except Exception as e:
        logging.error(f"Failed to send Telegram alert: {e}")

def try_restart_container(client: docker.DockerClient, container_name: str) -> bool:
    """Attempt to restart a stopped container."""
    try:
        container = client.containers.get(container_name)
        if container.status != "running":
            container.start()
            time.sleep(3)
            container.reload()
            return container.status == "running"
    except Exception as e:
        logging.error(f"Failed to restart {container_name}: {e}")
    return False

def handle_event(client: docker.DockerClient, event: dict):
    """Process a Docker event."""
    container_name = event["Actor"]["Attributes"].get("name", "unknown")
    event_type = event["Action"]
    timestamp = datetime.fromtimestamp(event["time"]).strftime("%H:%M:%S")

    # Skip containers we don't care about
    if container_name not in WATCHED_CONTAINERS:
```

```
        return

    if event_type in ALERT_EVENTS:
        exit_code = event["Actor"]["Attributes"].get("exitCode", "unknown")

        # Try auto-restart
        restarted = try_restart_container(client, container_name)

        if restarted:
            message = (
                f"Container `{container_name}` *died* (exit: {exit_code}) "
                f"at {timestamp}\n"
                f"  Auto-restarted successfully"
            )
            send_telegram(message, critical=False)
        else:
            message = (
                f"Container `{container_name}` *DIED* (exit: {exit_code}) "
                f"at {timestamp}\n"
                f"  Auto-restart FAILED - manual intervention required"
            )
            send_telegram(message, critical=True)

        logging.warning(f"Container {container_name} {event_type} (exit: {exit_code})")

    elif event_type in INFO_EVENTS:
        logging.info(f"Container {container_name} {event_type}")
        # Only notify on restarts (not initial starts)
        if event_type == "restart":
            message = f"Container `{container_name}` restarted at {timestamp}"
            send_telegram(message, critical=False)

def main():
    client = docker.from_env()
    logging.info("Docker events listener started")

    while True:
        try:
            for event in client.events(decode=True, filters={"type": "container"}):
                handle_event(client, event)
        except Exception as e:
            logging.error(f"Error in event loop: {e}")
            time.sleep(5)  # Brief pause before reconnecting

if __name__ == "__main__":
    main()
```

```
# /etc/systemd/system/docker-events-listener.service

[Unit]
Description=Docker Events Listener for Telegram Alerts
After=docker.service
Requires=docker.service

[Service]
Type=simple
User=hieudc
ExecStart=/home/hieudc/.venv/bin/python3 /home/hieudc/scripts/docker-events-listener.py
Restart=always
RestartSec=10
EnvironmentFile=/home/hieudc/.env

[Install]
WantedBy=multi-user.target
```

With this running, container failures generate Telegram alerts in 3-5 seconds, compared to 2-5 minutes through Prometheus + Alertmanager.

Agent Mesh Monitor: Cross-Server Health Checking

With agents running across multiple servers, you need a health checking system that can see across server boundaries. I built what I call an "agent mesh monitor" — a lightweight script that checks the health of every agent service from a central vantage point.

```
#!/bin/bash
# /home/hieudc/scripts/agent-mesh-monitor.sh
# Runs on Server-3 (monitoring server), checks all other servers

TELEGRAM_BOT_TOKEN="${TELEGRAM_BOT_TOKEN}"
TELEGRAM_CHAT_ID="${TELEGRAM_CHAT_ID}"
ALERT_COOLDOWN=300  # 5 minutes between repeated alerts
ALERT_STATE_FILE="/tmp/mesh-alert-state.json"

servers=(
    "server-1:10.10.0.2"
    "server-2:10.10.0.3"
    "server-3:10.10.0.1"
    "loadbalancer:10.10.0.4"
    "oracle:10.10.0.5"
)

# Services to check per server (name:port:path)
declare -A server_services
```

```
server_services["server-1"]="openclaw-gateway:3000:/health traefik:8080:/ping n8n:5678/ postgre
server_services["server-2"]="cognee-api:8000/health neo4j:7474/ postgres:5432"
server_services["server-3"]="prometheus:9090/-/healthy grafana:3000/api/health loki:3100/ready"
server_services["loadbalancer"]="traefik:8080:/ping haproxy:8404/stats"
server_services["oracle"]="cognee-mcp:8080/health"

check_http() {
    local host="$1"
    local port="$2"
    local path="${3:-/}"
    local timeout=5

    status=$(curl -s -o /dev/null -w "%{http_code}" \
        --connect-timeout "$timeout" \
        --max-time "$timeout" \
        "http://${host}:${port}${path}")

    [ "$status" = "200" ] || [ "$status" = "204" ]
}

check_tcp() {
    local host="$1"
    local port="$2"
    timeout 3 bash -c ">/dev/tcp/$host/$port" 2>/dev/null
}

send_alert() {
    local message="$1"
    local alert_key="$2"

    # Check cooldown
    if [ -f "$ALERT_STATE_FILE" ]; then
        last_alert=$(python3 -c "
import json, time
with open('$ALERT_STATE_FILE') as f:
    state = json.load(f)
print(state.get('$alert_key', 0))
" 2>/dev/null || echo 0)

        now=$(date +%s)
        if [ $((now - last_alert)) -lt $ALERT_COOLDOWN ]; then
            return  # Still in cooldown
        fi
    fi

    # Send alert
    curl -s -X POST \
```

```
        "https://api.telegram.org/bot${TELEGRAM_BOT_TOKEN}/sendMessage" \
        -d "chat_id=${TELEGRAM_CHAT_ID}&text=${message}&parse_mode=Markdown" > /dev/null

    # Update state
    python3 -c "
import json, time
try:
    with open('$ALERT_STATE_FILE') as f:
        state = json.load(f)
except:
    state = {}
state['$alert_key'] = int(time.time())
with open('$ALERT_STATE_FILE', 'w') as f:
    json.dump(state, f)
" 2>/dev/null
}

failures=0
report_lines=()

for server_entry in "${servers[@]}"; do
    server_name="${server_entry%%:*}"
    server_ip="${server_entry##*:}"

    # First check: is the server reachable at all?
    if ! ping -c 1 -W 2 "$server_ip" > /dev/null 2>&1; then
        msg="  *Server Unreachable*: ${server_name} (${server_ip})"
        send_alert "$msg" "server_down_${server_name}"
        report_lines+=("  ${server_name}: UNREACHABLE")
        ((failures++))
        continue
    fi

    report_lines+=("  ${server_name}: reachable")
done

# Send daily summary (not alerts)
if [ "${1}" = "--report" ]; then
    report=$(printf '%s\n' "${report_lines[@]}")
    curl -s -X POST \
        "https://api.telegram.org/bot${TELEGRAM_BOT_TOKEN}/sendMessage" \
        -d "chat_id=${TELEGRAM_CHAT_ID}&text=Mesh Health Report%0A${report}" > /dev/null
fi

exit $failures
```

Per-Server Health Check Scripts

Each server also runs its own local health check. Here's a simplified version showing the pattern:

Server-1: 22 services checked

```
#!/bin/bash
# /home/hieudc/scripts/server1-health-check.sh

check_docker_container() {
    local name="$1"
    status=$(docker inspect --format='{{.State.Status}}' "$name" 2>/dev/null)
    if [ "$status" = "running" ]; then
        echo "  $name"
        return 0
    else
        echo "  $name (status: ${status:-not found})"
        return 1
    fi
}

check_systemd_service() {
    local name="$1"
    if systemctl is-active --quiet "$name" 2>/dev/null; then
        echo "  $name"
        return 0
    else
        echo "  $name"
        return 1
    fi
}

# Docker services (14)
DOCKER_SERVICES=(
    "openclaw-gateway"
    "traefik"
    "socket-proxy"
    "n8n"
    "n8n-postgres"
    "chatwoot-web"
    "chatwoot-sidekiq"
    "chatwoot-postgres"
    "chatwoot-redis"
    "cliproxyapi"
    "antigravity-manager"
    "postgres-main"
    "redis-main"
    "pgbouncer"
)
```

```
# Systemd services (8)
SYSTEMD_SERVICES=(
    "openclaw"
    "docker"
    "wg-quick@wg-vn"
    "prometheus-node-exporter"
    "docker-events-listener"
    "health-check.timer"
    "sshd"
    "ufw"
)

echo "=== Server-1 Health Check $(date) ==="
echo ""
echo "--- Docker Services ---"
failures=0
for svc in "${DOCKER_SERVICES[@]}"; do
    check_docker_container "$svc" || ((failures++))
done

echo ""
echo "--- Systemd Services ---"
for svc in "${SYSTEMD_SERVICES[@]}"; do
    check_systemd_service "$svc" || ((failures++))
done

echo ""
echo "--- Summary ---"
echo "Failures: $failures / $((${#DOCKER_SERVICES[@]} + ${#SYSTEMD_SERVICES[@]}))"

exit $failures
```

War Story #1: The Wrong Service Check

Date: February 2026 **Impact:** False health reports for weeks **Symptom:** Health checks reported openclaw-gateway healthy when it was actually broken

What Happened

I had a health check script on Server-1 that checked if the `openclaw` systemd service was running. It looked fine:

```
check_systemd_service "openclaw"
#   openclaw
```

But OpenClaw kept behaving erratically. Telegram responses were intermittent. Sometimes tasks

would complete normally, sometimes they'd time out. The health check consistently reported green.

After digging in, I found the problem: there were two different `openclaw` services.

```
# User-level systemd service (the one that actually mattered)
systemctl --user status openclaw
#   openclaw.service - OpenClaw AI Agent
#     Loaded: loaded (/home/hieudc/.config/systemd/user/openclaw.service; enabled)
#     Active: failed (Result: exit-code) since ...

# System-level systemd service (what my check was hitting)
systemctl status openclaw
#   openclaw.service - OpenClaw Gateway (system stub)
#     Loaded: loaded (/etc/systemd/system/openclaw.service; enabled)
#     Active: active (running) since ...  ← This was just a stub/wrapper
```

My health check script used `systemctl is-active openclaw` without the `--user` flag. It was checking the system-level stub (which was just a symlink or wrapper), not the actual user-level service where OpenClaw actually ran.

The Fix

```
# WRONG: checks system-level service
systemctl is-active openclaw

# CORRECT: checks user-level service
systemctl --user is-active openclaw

# Even better: be explicit about what you're checking
check_user_service() {
    local name="$1"
    local user="$2"

    if sudo -u "$user" systemctl --user is-active --quiet "$name" 2>/dev/null; then
        echo "  [user:${user}] $name"
        return 0
    else
        echo "  [user:${user}] $name"
        return 1
    fi
}

check_user_service "openclaw" "hieudc"
```

Lesson Learned

When you have both system-level and user-level systemd services, your monitoring scripts need to be explicit about which scope they're checking. The default (`systemctl status name`) checks system scope. Add `--user` and optionally `-M user@` for user services.

Also: when something "passes health checks" but behaves erratically, the health check itself may be wrong. Question your monitoring before assuming the service is fine.

War Story #2: The 102-Restart Crash Loop

Date: February 2026 **Impact:** Gateway instability for hours **Symptom:** OpenClaw responding inconsistently, occasional duplicate messages

What Happened

I noticed OpenClaw was sending duplicate Telegram messages occasionally. Every few minutes, responses would come twice. Sometimes tasks would appear to run twice. Digging into the logs:

```
journalctl -u openclaw --no-pager -n 100 | grep "restart"
# Feb 07 21:30:01 systemd[1]: openclaw.service: Main process exited
# Feb 07 21:30:11 systemd[1]: openclaw.service: Start request repeated too quickly
# ... (repeated 102 times)
```

102 restart attempts. Something was causing the service to crash repeatedly, systemd kept restarting it, and each restart that survived long enough would pick up pending messages — but the previous crashed instance had already partially processed some of them.

Root Cause: Two Processes Running

```
pgrep -la "openclaw\|gateway"
# 768492 node /home/hieudc/.openclaw/gateway/index.js
# 771203 node /home/hieudc/.openclaw/gateway/index.js
```

Two gateway processes running simultaneously. Both were connected to the same Telegram webhook, both were processing incoming messages, and both were executing tasks. This caused: - Duplicate message processing - Race conditions on shared state files - Port conflicts causing crashes (one process binding a port the other was using) - The crash → restart → conflict → crash loop

How It Happened

A cron job that was supposed to ensure the gateway was running had a race condition:

```
# Broken cron (runs every minute)
if ! systemctl is-active --quiet openclaw; then
    systemctl start openclaw
fi
```

The check-and-start wasn't atomic. In a scenario where the service was starting (not yet "active" but not "failed"), the cron would see "not active" and issue another `start` — resulting in two starts, two processes, chaos.

The Fix

```
# Use systemctl's built-in restart logic instead of manual check-and-start
# This is atomic
systemctl restart openclaw

# Or use the idempotent ensure-running pattern:
ensure_single_instance() {
    local service="$1"

    # Kill any orphan processes first
    local pid_count
    pid_count=$(pgrep -c -f "openclaw/gateway" 2>/dev/null || echo 0)

    if [ "$pid_count" -gt 1 ]; then
        echo "WARNING: $pid_count instances found, killing extras"
        # Keep only the process managed by systemd
        local systemd_pid
        systemd_pid=$(systemctl show -p MainPID openclaw | cut -d= -f2)
        pgrep -f "openclaw/gateway" | grep -v "$systemd_pid" | xargs kill -TERM 2>/dev/null
        sleep 2
    fi

    systemctl is-active --quiet "$service" || systemctl start "$service"
}
```

And add a monitoring alert for multiple instances:

```
# prometheus rule
- alert: DuplicateAgentProcesses
  expr: |
    count by (instance) (
      process_cpu_seconds_total{job="openclaw"}
    ) > 1
  for: 1m
  labels:
    severity: critical
  annotations:
    summary: "Multiple OpenClaw processes detected"
    description: "{{ $value }} OpenClaw processes running on {{ $labels.instance }}. Expected
```

Lesson Learned

Duplicate process detection needs to be explicit in your monitoring. Standard health checks only verify "is something running" — they don't verify "is exactly one thing running." For agent services where duplicate execution causes real harm (duplicate API calls, race conditions, data corruption), add a check for exact process counts.

```
# Quick check: how many instances of your agent are running?
pgrep -c -f "your-agent-process-name"
# Should be exactly 1
```

Neural Memory Audit: When Agents Have Their Own Health

AI agents that use long-term memory systems (vector databases, knowledge graphs) need their own category of health monitoring. The memory system can degrade over time in ways that don't show up in standard infrastructure metrics.

I ran a formal audit of OpenClaw's neural memory (46MB, 297 memories at the time) and graded it:

```
Neural Memory Audit Report
Date: February 2026
Agent: OpenClaw

Criteria:
  Recency:      Grade B  (70%) - Memories from recent sessions sparse
  Coherence:    Grade D  (45%) - Many contradictory/stale entries
  Coverage:     Grade C  (60%) - Key operational knowledge missing
  Accessibility: Grade C (65%) - Query latency degrading
  Consolidation: Grade F (20%) - No deduplication done in 3+ weeks

Overall Grade: D (54.2/100)

Issues Found:
- 34 duplicate or near-duplicate memories
- 12 memories with contradictory information (e.g., two different server IPs for same service)
- 28 memories referencing services that no longer exist
- No consolidation run in 23 days

Recommendation: Emergency consolidation + pruning before memory system
becomes unreliable enough to impact agent behavior.
```

A grade D memory means the agent is working with stale, contradictory, and incomplete information. This doesn't cause an outage — the agent keeps running — but it causes subtle degradation in response quality and decision-making.

Memory Health Metrics to Track

```
# memory_health_exporter.py
from prometheus_client import Gauge, start_http_server
import sqlite3
import time
from datetime import datetime, timedelta
```

```
memory_age_days = Gauge(
    'agent_memory_oldest_entry_age_days',
    'Age of the oldest memory entry in days'
)

memory_total_count = Gauge(
    'agent_memory_total_count',
    'Total number of memory entries'
)

memory_duplicate_ratio = Gauge(
    'agent_memory_duplicate_ratio',
    'Estimated ratio of duplicate memories (0-1)'
)

memory_last_consolidation_days = Gauge(
    'agent_memory_last_consolidation_days',
    'Days since last memory consolidation run'
)

def update_memory_metrics(db_path: str):
    try:
        conn = sqlite3.connect(db_path)
        cursor = conn.cursor()

        # Total count
        cursor.execute("SELECT COUNT(*) FROM memories")
        count = cursor.fetchone()[0]
        memory_total_count.set(count)

        # Age of oldest entry
        cursor.execute("SELECT MIN(created_at) FROM memories")
        oldest = cursor.fetchone()[0]
        if oldest:
            oldest_dt = datetime.fromisoformat(oldest)
            age = (datetime.now() - oldest_dt).days
            memory_age_days.set(age)

        conn.close()
    except Exception as e:
        print(f"Error updating memory metrics: {e}")

if __name__ == "__main__":
    start_http_server(8092)
    while True:
        update_memory_metrics("/home/hieudc/.openclaw/memory.db")
        time.sleep(300)  # Update every 5 minutes
```

Grafana Dashboard Structure

My production Grafana setup has four dashboards:

Dashboard 1: Infrastructure Overview

```
Row 1: Server Status
  - CPU usage per server (multi-line time series)
  - Memory usage per server
  - Disk usage per server

Row 2: Network
  - Network I/O per server
  - WireGuard tunnel latency

Row 3: Docker
  - Container count per server
  - Restart rate (should be near zero)
  - Container CPU/memory by container name
```

Dashboard 2: AI Agent Metrics

```
Row 1: Health
  - Agent process count (alert if != 1)
  - Uptime percentage
  - Error rate (API failures / total requests)

Row 2: Performance
  - Request latency P50/P95/P99
  - Requests per minute
  - Queue depth (if applicable)

Row 3: Cost
  - Daily spend (USD, bar chart)
  - Token consumption rate
  - Cost by model breakdown
```

Dashboard 3: Logs (Loki)

```
Full-text log search
Filtered views:
  - Errors only
  - Agent task logs
  - API call logs
  - Security events (auth failures, rate limits)
```

Dashboard 4: Alerts History

```
- Recent alerts timeline
- Alert frequency by type
- MTTR (Mean Time to Recovery) by alert
- False positive rate
```

Log Aggregation with Loki

Centralized logging is essential when agents run across multiple servers. Without it, debugging requires SSH-ing to each server separately to read logs.

```yaml
# docker-compose.yml for Loki stack on Server-3

services:
  loki:
    image: grafana/loki:latest
    ports:
      - "10.10.0.1:3100:3100"
    volumes:
      - ./loki-config.yml:/etc/loki/config.yml
      - loki-data:/loki
    command: -config.file=/etc/loki/config.yml

  promtail:
    image: grafana/promtail:latest
    volumes:
      - /var/log:/var/log:ro
      - /var/lib/docker/containers:/var/lib/docker/containers:ro
      - ./promtail-config.yml:/etc/promtail/config.yml
    command: -config.file=/etc/promtail/config.yml
```

```yaml
# promtail-config.yml (runs on EACH server, ships logs to Loki)
server:
  http_listen_port: 9080

clients:
  - url: http://10.10.0.1:3100/loki/api/v1/push

scrape_configs:
  # Docker container logs
  - job_name: docker
    docker_sd_configs:
      - host: unix:///var/run/docker.sock
        refresh_interval: 5s
    relabel_configs:
      - source_labels: ['__meta_docker_container_name']
```

```
        target_label: container
      - source_labels: ['__meta_docker_container_label_com_docker_compose_service']
        target_label: service
    pipeline_stages:
      - docker: {}

  # Systemd journal logs
  - job_name: systemd
    journal:
      max_age: 12h
      labels:
        job: systemd
    relabel_configs:
      - source_labels: ['__journal__systemd_unit']
        target_label: unit
```

The Complete Monitoring Checklist

Use this when onboarding a new agent service:

```
Infrastructure Monitoring:
- [ ] node_exporter installed and scraping on VPN IP
- [ ] cadvisor installed for Docker metrics
- [ ] UFW rules allow Prometheus scrape from Server-3 (10.10.0.1)
- [ ] Added to prometheus.yml scrape_configs
- [ ] promtail installed, shipping logs to Loki

Agent-Specific:
- [ ] Health endpoint exists (/health or /ping)
- [ ] Prometheus metrics endpoint exists (/metrics)
- [ ] AI cost exporter configured if agent makes API calls
- [ ] Alert rules added to prometheus/rules/
- [ ] Added to docker-events-listener WATCHED_CONTAINERS set
- [ ] Added to per-server health check script
- [ ] Added to agent mesh monitor

Alerting:
- [ ] ContainerDown alert fires within 5 minutes of container stopping
- [ ] Test alert by stopping container and verifying Telegram notification
- [ ] Cooldown periods set to avoid alert storms
- [ ] Alert routes correctly to critical vs warning channels

Dashboards:
- [ ] Added to Infrastructure Overview dashboard
- [ ] Added to AI Agent Metrics dashboard (if applicable)
- [ ] Log queries work in Grafana/Loki
```

Summary

Monitoring AI agents requires going beyond traditional infrastructure metrics. You need to track:

1. **Health**: Is the agent running, and exactly one instance of it?
2. **Tokens**: Is consumption normal, or is something burning quota?
3. **Errors**: API failures, rate limits, context limits — all need visibility
4. **Latency**: Response time degradation is an early warning sign
5. **Cost**: Real-time spend tracking prevents surprise bills

The Docker events listener is the highest-value addition to this stack — under 5-second alert latency compared to minutes through Prometheus. If you implement nothing else from this chapter, implement that.

The war stories here share a common theme: standard monitoring reported green while something was actually wrong. The solution isn't better monitoring of the wrong things — it's monitoring the right things explicitly. Check for duplicate processes. Check the correct systemd scope. Question health checks that report healthy during erratic behavior.

Chapter 10 goes deeper into debugging — what to do once the alerts fire and you need to figure out what's actually wrong.

Chapter 10: Debugging AI Agents — When Things Go Wrong

> "Debugging a traditional service is like finding a lost key in your house. Debugging an AI agent is like finding a lost key in a house where the rooms rearrange themselves while you're looking."

Debugging AI agents is a genuinely different discipline from debugging traditional software. Traditional bugs are deterministic: given the same input, you get the same wrong output. You can reproduce them, step through them with a debugger, and isolate the exact line of code that's wrong.

Agent bugs are different. An agent's behavior depends on model responses, which are probabilistic. The same prompt can produce different behavior on different runs. The agent may have modified its own environment — changed a config file, moved a file, sent a message — making the reproduction context different from what it was when the bug occurred. And by the time you notice something is wrong, the agent may have taken dozens of actions you need to untangle.

This chapter catalogs the failure modes I've encountered running AI agents in production, with specific debugging approaches for each.

The Debugging Mindset Shift

Before jumping into specific failures, it's worth establishing what's different about agent debugging.

Traditional debugging assumptions that don't hold for agents:

"I can reproduce this reliably" — Not always. Model behavior varies. If the bug is in how the agent interprets an ambiguous situation, you may not be able to reliably trigger it again.

"The code is the source of truth" — Agents can modify their own operating environment. Configuration files, environment variables, state files — any of these may have been changed by the agent itself. What you see in the codebase may not reflect what was running when the bug occurred.

"Errors are explicit" — Agent failures are often silent. The agent finishes, reports success, and something is subtly wrong. Or it never reports at all.

"The system is stateless between runs" — Agents often maintain state across sessions: memory systems, context files, conversation history. A bug in one session can poison the state for future

sessions.

The Agent Debugging Stack

```
Level 5: Task-level behavior (did the agent accomplish the goal?)
Level 4: Tool execution (did individual tool calls succeed?)
Level 3: API calls (did model API requests/responses look correct?)
Level 2: Process health (is the agent process running correctly?)
Level 1: Infrastructure (are dependent services available?)
```

Work bottom-up. Verify infrastructure first, then process health, then API behavior, then tool execution, then task outcomes. Most engineers go straight to Level 5 and waste time debugging agent behavior when the actual problem is a Redis connection failure at Level 1.

Failure Mode 1: Agent Modifying Its Own Infrastructure

Frequency: Uncommon but catastrophic when it happens **Detection:** Service disruption immediately after agent task

The Incident

OpenClaw needed to configure its Claude Code integration to route through CLIProxyAPI. The task was: update the Claude Code configuration so it uses the proxy endpoint.

The agent looked at the available configuration files, found the CLIProxyAPI config at `~/.cli-proxy-api/config.json`, and modified it. This was the exact wrong file — it modified the proxy service's own configuration rather than the client that connects to the proxy.

```
What the agent should have done:
  Edit ~/.claude.json → set ANTHROPIC_BASE_URL to CLIProxyAPI endpoint

What the agent actually did:
  Edit ~/.cli-proxy-api/config.json → changed the proxy's internal routing
  (Breaking the proxy that the agent itself was running through)
```

The result: CLIProxyAPI was misconfigured, stopped routing requests correctly, and the agent lost its ability to make API calls — mid-task. The agent couldn't even report what had happened because the API it used for reporting was now broken.

Why It Happens

Agents make decisions based on file names, paths, and content. When an agent sees `config.json` in a directory called `cli-proxy-api`, it reasonably infers that's a configuration file it should edit. The agent doesn't inherently understand which services are "itself" versus "external services it uses."

Prevention

```
# In your agent's system prompt or AGENTS.md
```

```
## Infrastructure Constraints

The following services are part of the agent's own infrastructure.
NEVER modify their configuration files or restart them during operation:

- CLIProxyAPI (~/.cli-proxy-api/) - The API proxy you run through.
  Modifying this breaks your own API access.
- OpenClaw Gateway (~/.openclaw/gateway/) - Core agent runtime.
- systemd services: openclaw, cliproxyapi, docker

If a task requires modifying these, STOP and ask the user.
These changes must be made manually by the operator.
```

```
# Implement file write guards in your agent tooling
PROTECTED_PATHS = [
    "~/.cli-proxy-api/",
    "~/.openclaw/gateway/",
    "/etc/systemd/",
    "/etc/traefik/",
]

def safe_write_file(path: str, content: str) -> dict:
    """Write file with protection against modifying critical infrastructure."""
    import os
    abs_path = os.path.expanduser(os.path.abspath(path))

    for protected in PROTECTED_PATHS:
        protected_abs = os.path.expanduser(os.path.abspath(protected))
        if abs_path.startswith(protected_abs):
            return {
                "success": False,
                "error": (
                    f"BLOCKED: {path} is in a protected infrastructure directory. "
                    f"This operation requires manual operator intervention."
                )
            }

    # Proceed with write
    with open(abs_path, "w") as f:
        f.write(content)
    return {"success": True, "path": abs_path}
```

Recovery

```
# When an agent breaks its own infrastructure:

# 1. Find what changed recently
```

```
git -C ~/.cli-proxy-api diff 2>/dev/null || \
  find ~/.cli-proxy-api -newer /tmp/checkpoint -name "*.json" 2>/dev/null

# 2. Check service status
systemctl status cliproxyapi
journalctl -u cliproxyapi -n 50

# 3. Restore from backup or fix manually
# Never let the broken agent try to fix itself - it might make it worse

# 4. Verify the fix before restarting the agent
# Test the proxy directly:
curl -X POST http://localhost:8317/v1/messages \
  -H "Authorization: Bearer your-key" \
  -H "Content-Type: application/json" \
  -d '{"model": "claude-haiku-4", "max_tokens": 10, "messages": [{"role": "user", "content": "
```

Failure Mode 2: API Key Typos and Credential Errors

Frequency: Very common **Detection:** 401 Unauthorized errors, silent authentication failures

The Incident

OpenClaw was configured to use API key `openclaw-proxy-key-2026`. After a configuration update, it stopped working. The logs showed intermittent 401 errors, but only from some code paths — others seemed fine.

The root cause was a typo that had existed for days: `nclaw-proxy-key-2026` instead of `openclaw-proxy-key-2026`. The missing `ope` prefix.

```
# ~/.claude.json had:
{
  "apiKey": "nclaw-proxy-key-2026"
}

# Should have been:
{
  "apiKey": "openclaw-proxy-key-2026"
}
```

This particular typo was painful to catch because: 1. The key wasn't obviously wrong — it looked like a key name 2. Some code paths used environment variables (which had the correct key), while others used the config file (which had the typo) 3. The 401 errors appeared intermittently depending on which code path executed

Debugging Authentication Issues

```
#!/bin/bash
# auth-debug.sh - Systematically check all credential sources

echo "=== Credential Audit ==="

echo ""
echo "--- Environment Variables ---"
env | grep -E "API_KEY|TOKEN|SECRET|PASSWORD" | sed 's/=.*/=REDACTED/'

echo ""
echo "--- Claude Config ---"
if [ -f ~/.claude.json ]; then
    python3 -c "
import json
with open(os.path.expanduser('~/.claude.json')) as f:
    config = json.load(f)
# Show keys but redact values
for k, v in config.items():
    if any(word in k.lower() for word in ['key', 'token', 'secret', 'password']):
        print(f'  {k}: {str(v)[:8]}...[REDACTED]')
    else:
        print(f'  {k}: {v}')
"
fi

echo ""
echo "--- Test API Connectivity ---"
# Test each API endpoint with its configured key
curl -s -o /dev/null -w "CLIProxy: %{http_code}\n" \
    http://localhost:8317/v1/models \
    -H "Authorization: Bearer $(grep -o 'openclaw[^"]*' ~/.claude.json 2>/dev/null || echo 'KE
```

The Pattern: Credential Inconsistency Across Files

A related problem: credentials configured correctly in one place but stale/wrong in another.

```
# Find all places a specific key or token appears
grep -r "openclaw-proxy" ~ --include="*.json" --include="*.env" --include="*.yml" 2>/dev/null

# Check for common key file locations
for f in \
    ~/.claude.json \
    ~/.claude/settings.json \
    ~/.bashrc \
    ~/.profile \
    ~/.env \
```

```
    ~/*/. env \
    ~/*/.env; do
    if [ -f "$f" ] && grep -q "API_KEY\|TOKEN\|SECRET" "$f" 2>/dev/null; then
        echo "Found credentials in: $f"
    fi
done
```

Failure Mode 3: Permission Errors

Frequency: Common **Detection:** Error messages containing "Permission denied", "EACCES", "cannot write to"

Common Permission Errors in Agent Contexts

npm prefix permission error:

```
# Symptom
npm install -g @anthropic-ai/claude-code
# Error: EACCES: permission denied, mkdir '/usr/local/lib/node_modules'

# Diagnosis
npm config get prefix
# /usr/local  ← requires root

# Fix: redirect npm global to user directory
npm config set prefix '~/.npm-global'
echo 'export PATH=~/.npm-global/bin:$PATH' >> ~/.bashrc
source ~/.bashrc

# Now install works without sudo
npm install -g @anthropic-ai/claude-code
```

SSH key permission error:

```
# Symptom
ssh user@server-3
# Warning: Unprotected private key file!
# Permissions 0644 for '/tmp/server3-key' are too open.
# Bad permissions. Try removing permissions for user: ...

# Fix
chmod 600 /tmp/server3-key
ssh -i /tmp/server3-key user@server-3
```

Docker socket permission error:

```
# Symptom
docker ps
```

```
# Got permission denied while trying to connect to the Docker daemon socket

# Diagnosis
ls -la /var/run/docker.sock
# srw-rw---- 1 root docker

# Fix: add user to docker group (requires logout/login)
sudo usermod -aG docker $USER
newgrp docker  # apply without logout

# Verify
groups | grep docker
```

Agent-specific: writing to read-only mounted volume:

```
# Symptom
# Agent task: "Write output to /workspace/results.json"
# Error: Read-only file system

# Diagnosis
docker inspect agent-container | python3 -c "
import json, sys
config = json.load(sys.stdin)
for mount in config[0]['Mounts']:
    print(f'{mount[\"Source\"]} → {mount[\"Destination\"]} (RW: {mount[\"RW\"]})')
"

# Fix: check docker-compose volumes for :ro flag
volumes:
  - ./workspace:/workspace     # read-write (correct for output)
  - ./config:/config:ro        # read-only (correct for config)
```

Building a Permission Diagnostic Tool

```
#!/bin/bash
# /home/hieudc/scripts/check-agent-permissions.sh

AGENT_USER="${1:-hieudc}"
WORKSPACE="${2:-/home/hieudc/.openclaw/workspace}"

echo "=== Agent Permission Audit for $AGENT_USER ==="

# Check workspace writability
if [ -w "$WORKSPACE" ]; then
    echo "  Workspace writable: $WORKSPACE"
else
    echo "  Workspace NOT writable: $WORKSPACE"
```

```
    ls -la "$(dirname $WORKSPACE)"
fi

# Check Docker access
if docker ps > /dev/null 2>&1; then
    echo "  Docker access: OK"
else
    echo "  Docker access: FAILED"
    echo "    Groups: $(groups)"
fi

# Check key directories
for dir in \
    "$HOME/.ssh" \
    "$HOME/.config" \
    "$HOME/.npm-global" \
    "$HOME/.venv"; do
    if [ -d "$dir" ]; then
        perms=$(stat -c "%a" "$dir")
        echo "  $dir exists (permissions: $perms)"
    else
        echo "  $dir does not exist"
    fi
done

# Check SSH keys
for key in $HOME/.ssh/id_*; do
    if [ -f "$key" ] && [[ "$key" != *.pub ]]; then
        perms=$(stat -c "%a" "$key")
        if [ "$perms" = "600" ] || [ "$perms" = "400" ]; then
            echo "  SSH key $key: permissions OK ($perms)"
        else
            echo "  SSH key $key: bad permissions ($perms, should be 600)"
        fi
    fi
done
```

Failure Mode 4: Rate Limits

Frequency: Very common at scale **Detection:** HTTP 429 responses, "quota exceeded" errors

The Vertex AI 429 Problem

During a mass import to Cognee (186 pages of DevOps documentation), the import process started hitting Vertex AI rate limits. The embedding model (`textembedding-gecko`) had a default quota of 5 RPM — 5 requests per minute. The import script was hitting it at 10-20 RPM.

```
# Initial (naive) import script
for page in pages:
    embedding = get_embedding(page.content)  # Direct call, no rate limiting
    store_in_vector_db(embedding)
```

```
Results:
- First ~2 minutes: OK
- Minutes 3-5: intermittent 429s
- After minute 5: solid 429s, import stalled
- Cognee backend: timed out waiting for embeddings
- User-visible symptom: "import seems stuck, search returns nothing"
```

Rate Limit Handling Patterns

```
import time
import logging
from functools import wraps
from typing import TypeVar, Callable, Any

T = TypeVar('T')

class RateLimiter:
    """Token bucket rate limiter."""

    def __init__(self, requests_per_minute: int):
        self.rpm = requests_per_minute
        self.min_interval = 60.0 / requests_per_minute
        self.last_call = 0.0

    def wait(self):
        """Block until we can make the next request."""
        now = time.time()
        elapsed = now - self.last_call
        if elapsed < self.min_interval:
            sleep_time = self.min_interval - elapsed
            time.sleep(sleep_time)
        self.last_call = time.time()


def with_retry_on_rate_limit(
    max_retries: int = 5,
    base_delay: float = 60.0
):
    """Decorator for functions that may hit rate limits."""
    def decorator(func: Callable[..., T]) -> Callable[..., T]:
        @wraps(func)
        def wrapper(*args, **kwargs) -> T:
```

```
            for attempt in range(max_retries):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    error_str = str(e).lower()
                    if "429" in error_str or "quota" in error_str or "rate limit" in error_str
                        if attempt == max_retries - 1:
                            raise
                        # Exponential backoff for rate limits
                        delay = base_delay * (2 ** attempt)
                        logging.warning(
                            f"Rate limit hit (attempt {attempt+1}/{max_retries}). "
                            f"Waiting {delay:.0f}s before retry."
                        )
                        time.sleep(delay)
                    else:
                        raise  # Not a rate limit error, re-raise immediately
        return wrapper
    return decorator


# Usage
limiter = RateLimiter(requests_per_minute=4)  # Stay under 5 RPM quota

@with_retry_on_rate_limit(max_retries=3, base_delay=60.0)
def get_embedding_safe(text: str) -> list:
    limiter.wait()  # Enforce rate limit
    return embedding_client.embed(text)


# Batch processing with checkpoint/resume
def batch_embed_with_checkpoint(
    items: list,
    checkpoint_file: str = "/tmp/embed_checkpoint.json"
) -> list:
    """Process batch with ability to resume from checkpoint on failure."""
    import json, os

    # Load checkpoint
    processed = set()
    results = {}
    if os.path.exists(checkpoint_file):
        with open(checkpoint_file) as f:
            checkpoint = json.load(f)
            processed = set(checkpoint.get("processed", []))
            results = checkpoint.get("results", {})
        logging.info(f"Resuming from checkpoint: {len(processed)} already processed")
```

```
    for i, item in enumerate(items):
        item_id = str(i)
        if item_id in processed:
            continue  # Skip already processed

        try:
            embedding = get_embedding_safe(item)
            results[item_id] = embedding
            processed.add(item_id)

            # Save checkpoint every 10 items
            if len(processed) % 10 == 0:
                with open(checkpoint_file, "w") as f:
                    json.dump({"processed": list(processed), "results": results}, f)
                logging.info(f"Checkpoint saved: {len(processed)}/{len(items)}")

        except Exception as e:
            logging.error(f"Failed item {i}: {e}")
            # Save checkpoint before raising
            with open(checkpoint_file, "w") as f:
                json.dump({"processed": list(processed), "results": results}, f)
            raise

    return [results.get(str(i)) for i in range(len(items))]
```

CLIProxyAPI Auto-Switch on Quota Exhaustion

For free-tier Google AI accounts managed through CLIProxyAPI, quota exhaustion is expected behavior. The proxy handles it automatically, but you need to understand when you've exhausted all accounts:

```
# Check current quota status across all accounts
journalctl -u cliproxyapi --no-pager --since "1 week ago" | \
    grep -E "quota|limit|switch|exhausted" | tail -20

# Count usage per account this week
journalctl -u cliproxyapi --no-pager --since "1 week ago" | \
    grep "Use OAuth" | \
    awk '{print $NF}' | \
    sed 's/\.json//' | \
    sort | uniq -c | sort -rn

# When ALL accounts are exhausted, you'll see:
# "No available accounts with remaining quota"
# At this point, wait for weekly reset or use paid API
```

Failure Mode 5: Agent Not Reporting Progress

Frequency: Very common **Impact:** User frustration, task abandonment, misdiagnosed failures

The Problem

A long-running agent task takes 10-15 minutes. The user sees nothing for 10 minutes, assumes the agent is stuck or crashed, and either kills it or starts a duplicate task. When the original task completes 5 minutes later, you have duplicate results or a broken workflow.

This is a UX bug, not a technical bug — but it causes real technical problems downstream.

The Pattern That Prevents This

```
# Agent task runner with mandatory progress reporting
import asyncio
import time
from typing import AsyncGenerator, Optional
from dataclasses import dataclass

@dataclass
class ProgressUpdate:
    step: int
    total_steps: int
    message: str
    eta_seconds: Optional[float] = None

async def long_running_task_with_progress(
    items: list,
    notify: callable  # Function that sends progress to user
) -> list:
    """Example of a task that reports progress throughout."""

    total = len(items)
    results = []
    start_time = time.time()

    # Announce start immediately
    await notify(f"Starting task: processing {total} items. I'll update every few minutes.")

    for i, item in enumerate(items):
        # Process item
        result = await process_item(item)
        results.append(result)

        # Report progress at meaningful intervals
        # (every 10 items or every 2 minutes, whichever comes first)
        if (i + 1) % 10 == 0 or (time.time() - start_time) % 120 < 1:
```

```
            elapsed = time.time() - start_time
            rate = (i + 1) / elapsed  # items per second
            eta = (total - i - 1) / rate if rate > 0 else None

            progress_msg = (
                f"Progress: {i+1}/{total} items processed "
                f"({(i+1)/total*100:.0f}%)"
            )
            if eta:
                progress_msg += f"\nEstimated time remaining: {eta/60:.1f} minutes"

            await notify(progress_msg)

    # Announce completion
    elapsed = time.time() - start_time
    await notify(
        f"Task complete! Processed {total} items in {elapsed/60:.1f} minutes."
    )

    return results
```

For agent systems specifically, implement a "heartbeat" output that confirms the agent is still alive even when it has nothing new to report:

```
import threading
import time

class AgentHeartbeat:
    """Sends periodic 'still working' messages to prevent user concern."""

    def __init__(self, notify_func: callable, interval_seconds: int = 120):
        self.notify = notify_func
        self.interval = interval_seconds
        self.last_update = time.time()
        self._thread = None
        self._stop = threading.Event()

    def start(self, task_description: str):
        self._stop.clear()
        self._task = task_description
        self._thread = threading.Thread(target=self._heartbeat_loop, daemon=True)
        self._thread.start()

    def update(self, message: str):
        """Call this when you have a real update - resets the heartbeat timer."""
        self.last_update = time.time()
        self.notify(message)
```

```
    def stop(self):
        self._stop.set()

    def _heartbeat_loop(self):
        while not self._stop.wait(timeout=self.interval):
            since_update = time.time() - self.last_update
            if since_update >= self.interval:
                self.notify(
                    f"Still working on: {self._task}\n"
                    f"(running for {since_update/60:.0f} minutes)"
                )
                self.last_update = time.time()
```

Failure Mode 6: Context Limit Exceeded Mid-Task

Frequency: Common for long-running sub-agents **Detection:** Abrupt task termination, error containing "context_length_exceeded" or "200000"

The Problem

Claude's context window is 200K tokens. A sub-agent working on a large codebase, or a task that involves reading many files, can hit this limit mid-task. When it does, the task terminates abruptly — often without a useful error message passed back to the orchestrator.

```
Typical scenario:
1. Orchestrator spawns sub-agent: "Analyze all 89 TypeScript files in src/ and generate documen
2. Sub-agent starts reading files, building context
3. At file 60/89: context hits 190K tokens
4. Sub-agent tries to read file 61: error - context limit exceeded
5. Sub-agent crashes or returns incomplete results
6. Orchestrator receives no clear failure signal
7. Documentation is generated for only 60/89 files - silently incomplete
```

Prevention: Context Budget Management

```
# context_budget.py
import anthropic
from typing import Optional

# Approximate token counts (1 token  4 characters for English code)
def estimate_tokens(text: str) -> int:
    return len(text) // 4

class ContextBudgetManager:
    """Tracks context usage and prevents limit exhaustion."""
```

```
    MAX_CONTEXT = 200_000  # Claude's limit
    SAFETY_MARGIN = 0.85   # Use at most 85% of limit

    def __init__(self, task_description: str = ""):
        self.used_tokens = estimate_tokens(task_description)
        self.chunks: list = []

    @property
    def remaining(self) -> int:
        return int(self.MAX_CONTEXT * self.SAFETY_MARGIN) - self.used_tokens

    @property
    def usage_percent(self) -> float:
        return self.used_tokens / self.MAX_CONTEXT * 100

    def can_fit(self, content: str) -> bool:
        return estimate_tokens(content) <= self.remaining

    def add(self, content: str, label: str = "") -> bool:
        """Add content to context. Returns False if would exceed budget."""
        tokens = estimate_tokens(content)
        if tokens > self.remaining:
            return False
        self.used_tokens += tokens
        self.chunks.append({"label": label, "tokens": tokens})
        return True

    def summary(self) -> str:
        return (
            f"Context: {self.used_tokens:,}/{self.MAX_CONTEXT:,} tokens "
            f"({self.usage_percent:.1f}%) - {self.remaining:,} remaining"
        )


# Usage in a file-processing agent task
def process_files_with_budget(file_paths: list) -> dict:
    budget = ContextBudgetManager(task_description="Analyze TypeScript files and generate docs
    processed = []
    skipped = []

    for path in file_paths:
        content = open(path).read()

        if budget.can_fit(content):
            budget.add(content, label=path)
            processed.append(path)
        else:
```

```
            skipped.append(path)
            print(f"Skipping {path}: would exceed context budget")
            print(budget.summary())

    if skipped:
        print(f"\nWARNING: {len(skipped)} files skipped due to context limits:")
        for f in skipped:
            print(f"  - {f}")
        print("Run a second pass to process remaining files.")

    return {"processed": processed, "skipped": skipped}
```

The Chunking Pattern

For tasks that inherently require more context than fits in one window, use explicit chunking:

```
def analyze_large_codebase(file_paths: list, output_dir: str) -> None:
    """Process files in chunks, each chunk within context limits."""
    CHUNK_SIZE = 20  # Files per chunk (tune based on average file size)

    chunks = [file_paths[i:i+CHUNK_SIZE] for i in range(0, len(file_paths), CHUNK_SIZE)]
    print(f"Processing {len(file_paths)} files in {len(chunks)} chunks")

    all_results = []
    for chunk_num, chunk in enumerate(chunks, 1):
        print(f"Processing chunk {chunk_num}/{len(chunks)}: {len(chunk)} files")

        # Each chunk gets a fresh context window
        chunk_result = analyze_chunk(chunk)
        all_results.extend(chunk_result)

        # Save intermediate results (checkpoint pattern)
        save_checkpoint(all_results, f"{output_dir}/checkpoint-{chunk_num}.json")

    # Final synthesis pass (reads checkpoints, not raw files)
    synthesize_results(all_results, output_dir)
```

Failure Mode 7: Model Mismatch

Frequency: Uncommon but insidious **Detection:** Performance degradation, unexpected behavior, model name in API responses

The Incident

The CLIProxyAPI was configured to serve Claude Opus requests, but due to an account quota issue, it was transparently downgrading requests to Claude Sonnet. The configuration file said

`claude-opus-4-5-thinking`, the API responses reported `claude-sonnet-4-5-thinking`, and the agent was behaving accordingly — less capable than expected.

```
# Check what model the proxy is actually serving
journalctl -u cliproxyapi --no-pager -n 20 | grep "model"
# Use OAuth provider=antigravity ... model=claude-sonnet-4-5-thinking
# Expected: model=claude-opus-4-5-thinking
```

The mismatch happened because: 1. Opus has a higher quota consumption rate 2. The primary account hit its Opus quota limit 3. The proxy fell back to Sonnet without explicitly notifying the caller 4. The agent continued running, appearing to work but with degraded reasoning quality

Detecting Model Mismatches

```
def verify_model_in_response(response, expected_model: str) -> None:
    """Verify the model actually used matches what was requested."""
    actual_model = response.model

    # Normalize model names for comparison (proxies sometimes add suffixes)
    def normalize(m: str) -> str:
        return m.lower().replace("-latest", "").replace("-thinking", "")

    if normalize(actual_model) != normalize(expected_model):
        import logging
        logging.warning(
            f"Model mismatch: requested '{expected_model}', "
            f"got '{actual_model}'. "
            f"Check proxy configuration and quota status."
        )
        # For critical tasks, raise an error
        # raise ValueError(f"Model mismatch: {expected_model} vs {actual_model}")

# Usage
response = client.messages.create(
    model="claude-opus-4",
    messages=[...]
)
verify_model_in_response(response, "claude-opus-4")
```

War Story: SSR CSS Hash Mismatch

Date: February 2026 **Symptom:** Website showing broken styles after deployment **Time to diagnose:** 45 minutes (should have been 5)

What Happened

A Next.js application was deployed. The build ran successfully on Server-2. The app was accessible. But the styling looked completely broken — no CSS applied, just raw HTML.

The error in the browser console:

```
Prop className did not match.
Server: "styles__Container-abc123"
Client: "styles__Container-xyz789"
```

The CSS class hash mismatch is a classic Next.js hydration issue. But the cause wasn't in the code — it was in the deployment process.

The sequence of events: 1. New build generated on Server-2 (build ID: `abc123`) 2. Nginx was serving the old build (build ID: `xyz789`) from the previous deployment 3. Server-side rendering used the old server's build → generated class `xyz789` 4. Client downloaded the new JS bundle → expected class `abc123` 5. Mismatch on every page load

The Fix

```
# Wrong: just deploying new build without restarting the server
rsync -av dist/ /var/www/app/
# ← Old server process still running with old build in memory

# Correct: restart the server after every build deployment
rsync -av dist/ /var/www/app/
pm2 restart app-name
# or
systemctl restart app-name
# or for Docker:
docker-compose up -d --force-recreate app
```

The Lesson

Server-side rendering applications keep compiled assets in memory. A new deployment doesn't automatically reload them. This is obvious in retrospect but easy to miss when you're focused on the build process rather than the running server.

Add this to your deployment script:

```
#!/bin/bash
# deploy.sh - Always restart after build

set -e  # Fail on any error

echo "Building..."
npm run build

echo "Deploying..."
rsync -av --delete dist/ /var/www/app/
```

```
echo "Restarting server..."
pm2 restart app-name

echo "Verifying..."
sleep 3
if curl -s -o /dev/null -w "%{http_code}" http://localhost:3000 | grep -q "200"; then
    echo "  Deployment successful"
else
    echo "  Server not responding after restart"
    pm2 logs app-name --lines 20
    exit 1
fi
```

War Story: Config Sync Across Three Servers

Date: February 2026 **Symptom:** Inconsistent agent behavior depending on which server handled the request **Time to diagnose:** Several days of intermittent complaints

What Happened

OpenClaw was updated to use a new model configuration. The update was applied to Server-1. Server-2 and Server-3 still had the old configuration. Since requests could be handled by any server through the load balancer, behavior was inconsistent — sometimes the new model, sometimes the old.

```
Request routing (round-robin):
  Request 1 → Server-1 → New model → Good response
  Request 2 → Server-2 → Old model → Different behavior
  Request 3 → Server-3 → Old model → Different behavior
  Request 4 → Server-1 → New model → Good response
  ...
```

The debugging challenge: inconsistent behavior looks like a flaky model or a bug in the agent logic. It took several days to realize the issue was configuration drift across servers.

The Diagnosis

```
# Compare config files across all servers
for server in server1 server2 server3; do
    echo "=== $server ==="
    ssh $server "cat ~/.openclaw/openclaw.json | python3 -m json.tool | head -20"
done

# Find differences
diff \
```

```
    <(ssh server1 "cat ~/.openclaw/openclaw.json | python3 -m json.tool") \
    <(ssh server2 "cat ~/.openclaw/openclaw.json | python3 -m json.tool")
```

Prevention: Config Sync Script

```
#!/bin/bash
# /home/hieudc/scripts/sync-agent-config.sh
# Run after any configuration change

SERVERS=("10.10.0.2" "10.10.0.3" "10.10.0.1")  # Server-1, 2, 3
SERVER_NAMES=("server-1" "server-2" "server-3")
SOURCE_SERVER="10.10.0.2"  # Server-1 is source of truth
CONFIG_FILES=(
    "~/.openclaw/openclaw.json"
    "~/.claude/settings.json"
    "~/.cli-proxy-api/config.json"
)

echo "=== Config Sync ==="
echo "Source: server-1 (${SOURCE_SERVER})"

for i in "${!SERVERS[@]}"; do
    target_ip="${SERVERS[$i]}"
    target_name="${SERVER_NAMES[$i]}"

    if [ "$target_ip" = "$SOURCE_SERVER" ]; then
        continue  # Skip source server
    fi

    echo ""
    echo "Syncing to ${target_name} (${target_ip})..."

    for config_file in "${CONFIG_FILES[@]}"; do
        # Expand ~ for rsync
        expanded_path="${config_file/#\~/$HOME}"

        if [ -f "$expanded_path" ]; then
            rsync -av "$expanded_path" "hieudc@${target_ip}:${config_file}" && \
                echo "    ${config_file}" || \
                echo "    Failed: ${config_file}"
        else
            echo "    Not found locally: ${config_file}"
        fi
    done

    # Restart affected services on target
    echo "  Restarting services on ${target_name}..."
```

```
    ssh "hieudc@${target_ip}" "systemctl --user restart openclaw 2>/dev/null; \
        docker restart cliproxyapi 2>/dev/null || true"
done

echo ""
echo "=== Verifying Config Consistency ==="
for i in "${!SERVERS[@]}"; do
    server_ip="${SERVERS[$i]}"
    server_name="${SERVER_NAMES[$i]}"
    checksum=$(ssh "hieudc@${server_ip}" "md5sum ~/.openclaw/openclaw.json 2>/dev/null" | cut
    echo "${server_name}: ${checksum}"
done
```

The Better Long-Term Solution: Centralized Config

For multi-server agent deployments, store configuration in a centralized store rather than syncing files manually:

```
# config_service.py - Centralized config with etcd or Consul
import etcd3
import json

class CentralizedConfig:
    def __init__(self, etcd_host: str = "10.10.0.2", etcd_port: int = 2379):
        self.client = etcd3.client(host=etcd_host, port=etcd_port)

    def get(self, key: str, default=None):
        value, _ = self.client.get(f"/agent/config/{key}")
        if value is None:
            return default
        return json.loads(value.decode())

    def set(self, key: str, value):
        self.client.put(f"/agent/config/{key}", json.dumps(value).encode())

    def watch(self, key: str, callback):
        """Watch for config changes and trigger callback."""
        events_iterator, cancel = self.client.watch_prefix(f"/agent/config/{key}")
        for event in events_iterator:
            callback(json.loads(event.value.decode()))

# All servers use the same source of truth
config = CentralizedConfig()
model = config.get("model", "claude-sonnet-4-6")
```

War Story: The Etcd Split-Brain

Date: February 2026 **Symptom:** 22 failover events in one day, inconsistent service behavior **Time to resolve:** 4 hours

What Happened

An etcd cluster (3 nodes) running across the servers developed a configuration mismatch. Server-1 had a different cluster ID than Servers 2 and 3. This caused a split-brain condition: the cluster couldn't agree on a leader, resulting in constant leader elections and failovers.

```
# Symptoms in etcd logs
etcdctl endpoint status --cluster
# Error: context deadline exceeded (from endpoints: ...)

# Member list showed the problem
etcdctl member list
# 8e9e05c5f3bf2c7a: ...  Server-1 (different cluster)
# ade526d28b1f92f3: ...  Server-2
# b61aea4a43e20cdb: ...  Server-3
```

The Fix

```
# Step 1: Identify the correct cluster members
etcdctl --endpoints=http://10.10.0.3:2379 member list  # Server-2 view
etcdctl --endpoints=http://10.10.0.1:2379 member list  # Server-3 view

# Step 2: Remove the diverged member
etcdctl member remove 8e9e05c5f3bf2c7a

# Step 3: Stop etcd on Server-1
systemctl stop etcd

# Step 4: Clear Server-1's data directory
rm -rf /var/lib/etcd/data/*

# Step 5: Add Server-1 back as a new member
etcdctl member add server-1 --peer-urls=http://10.10.0.2:2380

# Step 6: Start etcd on Server-1 with join parameters
cat > /etc/etcd/etcd.conf << EOF
ETCD_NAME="server-1"
ETCD_DATA_DIR="/var/lib/etcd/data"
ETCD_INITIAL_CLUSTER_STATE="existing"
ETCD_INITIAL_CLUSTER="server-1=http://10.10.0.2:2380,server-2=http://10.10.0.3:2380,server-3=ht
ETCD_LISTEN_PEER_URLS="http://10.10.0.2:2380"
ETCD_LISTEN_CLIENT_URLS="http://10.10.0.2:2379,http://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="http://10.10.0.2:2379"
```

```
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://10.10.0.2:2380"
EOF

systemctl start etcd

# Step 7: Verify cluster health
etcdctl endpoint health --cluster
# http://10.10.0.2:2379 is healthy
# http://10.10.0.3:2379 is healthy
# http://10.10.0.1:2379 is healthy
```

War Story: The Reasoning Parameter Bug

Date: February 2026 **Symptom:** Google API calls returning HTTP 400 errors **Time to diagnose:** ~2 hours

What Happened

After updating the agent configuration to use extended thinking (which uses a `reasoning_effort` parameter for Claude), the agent started failing on requests routed to Google Gemini models. The error was cryptic:

```
HTTP 400 Bad Request
{
  "error": {
    "code": 400,
    "message": "Invalid JSON payload received. Unknown name 'reasoning_effort'",
    "status": "INVALID_ARGUMENT"
  }
}
```

The agent framework was passing `reasoning_effort` to every model request, because it was set globally in the configuration. Claude understands `reasoning_effort`. Google Gemini does not — it returns a 400 error when it receives unknown parameters.

```
# Wrong: global params passed to all providers
def call_model(model: str, messages: list, **kwargs):
    return client.messages.create(
        model=model,
        messages=messages,
        reasoning_effort="high",  # ← Sent to Google, which doesn't support it
        **kwargs
    )
```

The Fix

```
# Provider-aware parameter filtering
PROVIDER_PARAMS = {
    "anthropic": {"reasoning_effort", "thinking", "betas"},
    "google": {"safety_settings", "generation_config"},
    "openai": {"logprobs", "top_logprobs", "seed"},
}

UNIVERSAL_PARAMS = {"max_tokens", "temperature", "top_p", "stop", "stream"}

def get_provider(model: str) -> str:
    if "claude" in model.lower():
        return "anthropic"
    elif "gemini" in model.lower() or "google" in model.lower():
        return "google"
    elif "gpt" in model.lower() or "o1" in model.lower():
        return "openai"
    return "unknown"

def call_model_safely(model: str, messages: list, **all_params) -> any:
    """Call model API with only parameters that provider supports."""
    provider = get_provider(model)
    allowed_params = UNIVERSAL_PARAMS | PROVIDER_PARAMS.get(provider, set())

    # Filter to only supported params
    filtered_params = {
        k: v for k, v in all_params.items()
        if k in allowed_params
    }

    # Log if params were filtered
    filtered_out = set(all_params.keys()) - set(filtered_params.keys())
    if filtered_out:
        import logging
        logging.debug(
            f"Filtered params for {provider}/{model}: {filtered_out}"
        )

    return client.messages.create(
        model=model,
        messages=messages,
        **filtered_params
    )
```

War Story: Claude Code Stuck at Trust Dialog

Date: February 2026 **Symptom:** Claude Code session unresponsive, tasks not executing **Time to diagnose:** 30 minutes

What Happened

Claude Code was running in a tmux session (`claude-code`), configured to automatically handle tasks from OpenClaw. Tasks were being sent to the session but getting no response. The session appeared active but idle.

```
# Capture what's actually showing in the tmux session
tmux capture-pane -p -t claude-code
# OUTPUT:
#
#                      Trust this folder?
#
#    /home/hieudc
#
#    Do you trust the files in this folder?
#
#    1. Yes, trust /home/hieudc and all subfolders
#    2. Yes, trust /home/hieudc (this session only)
#    3. No (recommended)
#
#
```

Claude Code had opened to `/home/hieudc` directory (which hadn't been trusted before) and was waiting for user input on a trust prompt. Because it was running in an automated tmux session, there was no human to click "Yes." All incoming tasks were queuing up behind this unanswered dialog.

The Fix (Short-term)

```
# Send "1" keypress to tmux session to accept trust prompt
tmux send-keys -t claude-code "1" Enter

# Verify it worked
sleep 2
tmux capture-pane -p -t claude-code | tail -5
```

The Fix (Long-term)

```
// ~/.claude.json - Pre-trust directories that Claude Code will work in
{
  "projects": {
    "/home/hieudc": {
      "hasTrustDialogAccepted": true
    },
```

```
    "/home/hieudc/.openclaw/workspace": {
      "hasTrustDialogAccepted": true
    },
    "/home/hieudc/projects": {
      "hasTrustDialogAccepted": true
    }
  }
}
```

Add trust dialog check to your session health monitor:

```
# Check if Claude Code is stuck on a dialog
check_claude_code_health() {
    local session="${1:-claude-code}"

    if ! tmux has-session -t "$session" 2>/dev/null; then
        echo "  Claude Code session not found: $session"
        return 1
    fi

    local content
    content=$(tmux capture-pane -p -t "$session" 2>/dev/null)

    if echo "$content" | grep -q "Trust this folder"; then
        echo "  Claude Code stuck on trust dialog"
        # Auto-accept (only safe if you trust the folder)
        tmux send-keys -t "$session" "1" Enter
        echo "  Auto-accepted trust dialog"
        return 0
    fi

    if echo "$content" | grep -q "Do you want to proceed"; then
        echo "  Claude Code waiting for confirmation"
        return 1  # Don't auto-accept arbitrary confirmations
    fi

    echo "  Claude Code session healthy"
    return 0
}
```

War Story: Sub-Agent Silent Failure

Date: February 2026 **Symptom:** Task reported as complete, directory didn't exist **Time to diagnose:** 45 minutes

What Happened

OpenClaw spawned a sub-agent (`Deploy_Expert`) to deploy Firecrawl on Server-3. The sub-agent ran, appeared to complete, and reported success. When I later tried to access Firecrawl at its expected location:

```
ls /root/firecrawl
# ls: cannot access '/root/firecrawl': No such file or directory
```

The directory didn't exist. The deployment had never happened. But the sub-agent had reported success.

What Actually Happened

The sub-agent had SSH'd to Server-3, run a few commands, encountered an error (likely a permission or network issue), and then — crucially — failed to propagate that error back up to the orchestrating agent. It returned a success message based on the commands it had successfully run, not on whether the deployment had actually worked.

```
Sub-agent execution (reconstructed):
    SSH to server-3 established
    apt-get update
    apt-get install docker-compose
    git clone https://github.com/mendableai/firecrawl /root/firecrawl
       Error: Could not resolve host: github.com (DNS issue on server-3)
  [Sub-agent continues with remaining commands on empty directory]
    "Setup complete" - reported to orchestrator
```

Prevention: Verification Steps in Sub-Agent Tasks

```
# In your sub-agent prompts, always include verification:

After completing the deployment:
1. Verify the application directory exists: `ls -la /root/firecrawl/`
2. Verify containers are running: `docker ps | grep firecrawl`
3. Test the API endpoint: `curl http://localhost:3002/v1/health`
4. Report the ACTUAL status - not what you expected, but what you verified.

If any verification step fails, report FAILURE with:
- Which step failed
- The exact error message
- What state the system is in now
Do NOT report success if verification fails.
```

```
# Orchestrator-side verification
async def deploy_with_verification(target_server: str, service_name: str) -> dict:
    """Deploy a service and verify it's actually running."""

    # Run the deployment
    deploy_result = await spawn_agent(
```

```
        task=f"Deploy {service_name} on {target_server}",
        verification_required=True
    )

    # Don't trust the agent's self-report - verify independently
    if deploy_result.get("success"):
        # Verify from a different angle (not the agent's perspective)
        verified = await verify_service_independently(target_server, service_name)

        if not verified:
            return {
                "success": False,
                "error": "Agent reported success but independent verification failed",
                "agent_report": deploy_result
            }

    return deploy_result


async def verify_service_independently(server: str, service: str) -> bool:
    """Verify service is running without trusting the deploying agent."""
    import asyncio
    try:
        result = await asyncio.create_subprocess_exec(
            "ssh", f"hieudc@{server}",
            f"docker ps --filter name={service} --format '{{{{.Status}}}}' | grep -q 'Up'",
            stdout=asyncio.subprocess.PIPE,
            stderr=asyncio.subprocess.PIPE
        )
        await result.communicate()
        return result.returncode == 0
    except Exception:
        return False
```

The Debugging Checklist

When something goes wrong with an AI agent, work through this checklist in order:

```
## Level 1: Infrastructure
- [ ] Can you reach the server? (ping, SSH)
- [ ] Are dependent services running? (DB, Redis, API proxy)
- [ ] Is disk space OK? (df -h)
- [ ] Is memory OK? (free -h)

## Level 2: Process Health
- [ ] Is the agent process running? (ps, systemctl status, docker ps)
```

```
- [ ] Is exactly ONE instance running? (pgrep -c)
- [ ] Are there crash loops? (journalctl, docker logs)
- [ ] Check resource limits not exceeded (memory, CPU quota)

## Level 3: API Connectivity
- [ ] Can the agent reach the model API? (test with curl)
- [ ] Is the API key correct and not expired?
- [ ] Is the proxy/CLIProxyAPI working?
- [ ] Are you hitting rate limits? (check for 429s)

## Level 4: Configuration
- [ ] Is config the same across all servers? (diff config files)
- [ ] Are you checking the right systemd scope? (--user vs system)
- [ ] Are environment variables set correctly?
- [ ] Did you restart the service after config changes?

## Level 5: Task-Level
- [ ] Is the agent reporting progress? (check logs)
- [ ] Did sub-agents independently verify their work?
- [ ] Is there a duplicate process causing conflicts?
- [ ] Is the model the agent is actually using what's configured?
- [ ] Is context approaching the 200K limit?
- [ ] Are there orphan tmux sessions consuming quota?
```

Summary

Agent debugging requires a fundamentally different approach than traditional software debugging. The key principles:

1. **Work bottom-up** — verify infrastructure before questioning agent behavior
2. **Don't trust agent self-reports** — verify outcomes independently, especially for deployments
3. **Explicit progress reporting is mandatory** — silence looks like failure to users
4. **Check for duplicate processes explicitly** — standard health checks won't catch them
5. **Scope your systemd checks** — user-level vs system-level services are different
6. **Filter provider-specific parameters** — what Claude accepts, Gemini may not
7. **Config drift across servers causes intermittent bugs** — check all servers, not just one
8. **Monitor tmux sessions** — orphan sessions silently burn quota and may run duplicate work
9. **Context limits cause silent truncation** — implement budget management before hitting the wall
10. **Protect critical infrastructure from agents** — agents need explicit constraints on what they can modify

The common thread in all these war stories: the agent was running, technically. It was doing things. The things it was doing were just wrong, in ways that standard "is it up?" monitoring couldn't catch.

That's why the monitoring chapter came before this one. You need observability in place before you need to debug. By the time something breaks, you want logs, metrics, and alert history ready to tell you what happened — not just what's happening now.

The agents will do things you didn't expect. Build the systems to catch it.

PART 4: ADVANCED TOPICS

Chapter 11: Security Hardening for AI Agent Infrastructure

> "The agent helpfully printed our Vault root token right there in the chat. Thirty seconds later I was frantically revoking it while drafting an incident report."

The Day the AI Became Our Biggest Security Hole

It started as a debugging session.

We had a new engineer onboarding, and I was showing him how our AI agent could help with infrastructure tasks. We asked the agent to help diagnose a Vault connectivity issue. The agent — trying to be helpful — asked us to paste the current environment variables so it could analyze the configuration.

We did.

The root token appeared in the chat history. Unredacted. In a conversation that synced to the cloud. In a log file that gets rotated but not encrypted. In a session that three other people had access to.

That was the moment we learned that AI agents in production infrastructure require a fundamentally different security model than traditional tools. Your engineers know not to paste credentials into Slack. But do they know not to share environment context with an AI agent that logs everything?

This chapter covers what we learned the hard way: how to harden AI agent infrastructure, lock down the surrounding services, and build a security posture that treats the AI as a potential attack vector, not just a productivity tool.

The Threat Model Has Changed

Traditional infrastructure security assumes humans make mistakes. You build guardrails — mandatory MFA, least-privilege IAM roles, secret scanning in CI/CD — and you train your team.

AI agents break this model in several ways:

Agents aggregate context. A human engineer might not realize they have access to both the database password and the production server list. An agent working a task will naturally pull both pieces of information together, increasing the blast radius of any compromise.

Agents surface hidden information. The agent asked what environment variables were set. No human would think to ask that in a debugging session. The agent did, because it was the logical next step. The answer included `VAULT_TOKEN=s.xxxxxxxx`.

Agents leave traces. Every conversation, every tool call, every intermediate result gets logged somewhere. In our setup, that meant chat history in the application database, tool call logs in the filesystem, and API request logs at the provider. Three places a leaked credential might live.

Agents can be prompted. An attacker who can influence what the agent works on — through a malicious issue, a crafted config file, a compromised input source — can potentially direct the agent to exfiltrate data or perform destructive actions.

Let's go through the hardening steps we implemented after our wake-up call.

Part 1: Vault Token Management

The Root Token Problem

Vault root tokens should never be used in production. Full stop. The root token bypasses all policies — it can do anything. We had been using it during initial setup and, through a series of "temporary" decisions, it had stayed in our environment variables for six months.

When the agent exposed it, we had to treat it as compromised.

Immediate remediation:

```
# Revoke the root token immediately
vault token revoke $EXPOSED_ROOT_TOKEN

# Generate a new root token ONLY if needed for emergency admin
# (requires unseal key holders to be present)
vault operator generate-root -init

# Distribute new unseal keys if rotation is needed
vault operator rekey -init -key-shares=5 -key-threshold=3
```

The right way to use Vault in production:

Never use root tokens for operational work. Create AppRoles with minimal policies.

```
# vault-policy-agent.hcl
# Policy for the AI agent's service account

path "secret/data/app/*" {
  capabilities = ["read"]
}
```

```
path "secret/data/infrastructure/database" {
  capabilities = ["read"]
}

# Explicitly deny access to sensitive paths
path "auth/*" {
  capabilities = ["deny"]
}

path "sys/*" {
  capabilities = ["deny"]
}

path "secret/data/vault-admin/*" {
  capabilities = ["deny"]
}
```

```
# Apply the policy
vault policy write agent-policy vault-policy-agent.hcl

# Create an AppRole for the agent
vault auth enable approle

vault write auth/approle/role/ai-agent \
  token_policies="agent-policy" \
  token_ttl=1h \
  token_max_ttl=4h \
  secret_id_ttl=24h

# Fetch the role ID (not secret - this is safe to store)
vault read auth/approle/role/ai-agent/role-id

# Generate a secret ID (store this securely, rotate regularly)
vault write -f auth/approle/role/ai-agent/secret-id
```

The agent gets a short-lived token. If it appears in a chat log, it's expired before anyone can use it.

Token Rotation Automation

```
#!/bin/bash
# rotate-vault-token.sh
# Run via cron or after any suspected exposure

set -euo pipefail

ROLE_ID=$(cat /etc/agent/vault-role-id)
SECRET_ID=$(cat /etc/agent/vault-secret-id)
```

```
# Login and get new token
NEW_TOKEN=$(vault write -field=token auth/approle/login \
  role_id="$ROLE_ID" \
  secret_id="$SECRET_ID")

# Update the running service
echo "VAULT_TOKEN=$NEW_TOKEN" > /etc/agent/vault-token
chmod 600 /etc/agent/vault-token

# Signal the agent to reload (implementation-specific)
systemctl reload ai-agent

# Revoke the old token (optional - it will expire anyway, but belt-and-suspenders)
if [ -n "${OLD_TOKEN:-}" ]; then
  VAULT_TOKEN=$NEW_TOKEN vault token revoke "$OLD_TOKEN"
fi

echo "Token rotation complete at $(date)"
```

Vault Seal/Unseal Procedures

Vault in production should be auto-unsealed using a cloud KMS (AWS KMS, GCP Cloud KMS, Azure Key Vault) or initialized with Shamir secret sharing across multiple key holders.

We use Shamir with five key holders, requiring three to unseal:

```
# Initialize Vault (only done once)
vault operator init \
  -key-shares=5 \
  -key-threshold=3 \
  -format=json > vault-init.json

# IMMEDIATELY distribute keys to different people/systems
# DO NOT store vault-init.json anywhere
cat vault-init.json | jq -r '.unseal_keys_b64[]'  # Distribute these
cat vault-init.json | jq -r '.root_token'          # Use once then revoke

# Unseal procedure (requires 3 of 5 key holders)
vault operator unseal  # Enter key 1
vault operator unseal  # Enter key 2
vault operator unseal  # Enter key 3

# Check seal status
vault status
```

For automated unsealing with AWS KMS:

```
# vault.hcl
seal "awskms" {
```

```
  region     = "ap-southeast-2"
  kms_key_id = "arn:aws:kms:ap-southeast-2:123456789:key/your-key-id"
}
```

Part 2: Network Hardening — Ports That Should Never Be Public

The Discovery

During our security audit, we ran a port scan from an external IP against our servers.

```
nmap -sV -p 1-65535 --open your-server-ip
```

The results were horrifying. Port 9100 (Prometheus Node Exporter), 9187 (PostgreSQL Exporter), 9081 (some internal service), 5432 (PostgreSQL itself on server-3), and 6379 (Redis) were all reachable from the public internet.

Someone could scrape our metrics — CPU, memory, disk, query statistics, slow query logs. Someone could attempt to connect directly to PostgreSQL. Someone could connect to Redis without authentication (we had `requirepass` set, but still).

Server-3 had UFW installed but **never enabled**. It had been provisioned six months ago and nobody had checked.

```
# The horrifying discovery
ssh server-3
sudo ufw status
# Status: inactive
```

Enabling UFW From Scratch (Without Locking Yourself Out)

This is the sequence that will save you from the classic "I enabled UFW and now I can't SSH in" disaster.

```
#!/bin/bash
# harden-ufw.sh
# Run this script to configure and enable UFW safely
# ALWAYS ensure SSH is allowed before enabling

# Reset to defaults (careful on live systems)
# sudo ufw reset

# Set default policies
sudo ufw default deny incoming
sudo ufw default allow outgoing

# SSH - MUST come first, before enabling
# Replace with your actual SSH port
sudo ufw allow from YOUR_VPN_CIDR to any port 22 proto tcp comment "SSH via VPN"
```

```
# If you need SSH from anywhere temporarily during setup:
# sudo ufw allow 22/tcp comment "SSH - TEMPORARY"

# Web traffic
sudo ufw allow 80/tcp comment "HTTP"
sudo ufw allow 443/tcp comment "HTTPS"

# Internal services - VPN only
VPN_CIDR="10.8.0.0/24"  # Adjust to your VPN subnet

# PostgreSQL - only from app servers via VPN
sudo ufw allow from "$VPN_CIDR" to any port 5432 proto tcp comment "PostgreSQL via VPN"

# Redis - only from app servers via VPN
sudo ufw allow from "$VPN_CIDR" to any port 6379 proto tcp comment "Redis via VPN"

# Prometheus exporters - only from monitoring server via VPN
MONITORING_IP="10.8.0.5"  # Your Prometheus server
sudo ufw allow from "$MONITORING_IP" to any port 9100 proto tcp comment "Node Exporter"
sudo ufw allow from "$MONITORING_IP" to any port 9187 proto tcp comment "PG Exporter"
sudo ufw allow from "$MONITORING_IP" to any port 9081 proto tcp comment "App metrics"

# Patroni (PostgreSQL HA) - only between DB nodes
DB_NODE1="10.8.0.10"
DB_NODE2="10.8.0.11"
DB_NODE3="10.8.0.12"
sudo ufw allow from "$DB_NODE1" to any port 8008 proto tcp comment "Patroni API"
sudo ufw allow from "$DB_NODE2" to any port 8008 proto tcp comment "Patroni API"
sudo ufw allow from "$DB_NODE3" to any port 8008 proto tcp comment "Patroni API"

# Verify rules before enabling
sudo ufw show added

# Enable (the scary part)
sudo ufw --force enable
sudo ufw status verbose
```

Binding Sensitive Services to VPN Interface

UFW rules are one layer. Binding services to the VPN interface is another, better, layer.

For PostgreSQL in `postgresql.conf`:

```
# Only listen on VPN interface and localhost - never on public interface
listen_addresses = '127.0.0.1,10.8.0.10'
```

For Redis in `redis.conf`:

```
bind 127.0.0.1 10.8.0.10
```

For Prometheus Node Exporter in systemd:

```
[Service]
ExecStart=/usr/local/bin/node_exporter \
  --web.listen-address="10.8.0.10:9100"
```

If the service only listens on the VPN interface, a firewall misconfiguration can't expose it. Defense in depth.

Oracle Server: NFS/rpcbind Cleanup

Our Oracle database server had `rpcbind` and NFS services running — remnants of a configuration that was copy-pasted from another server that needed NFS mounts. We hadn't needed NFS here in over a year.

```
# Check what's actually listening
ss -tlnp | grep -E '(111|2049|20048)'

# Stop and disable the services
sudo systemctl stop nfs-server rpcbind nfs-mountd
sudo systemctl disable nfs-server rpcbind nfs-mountd

# Verify they're gone
sudo systemctl is-enabled nfs-server rpcbind  # Should return "disabled"

# Update firewall to explicitly block these ports even if services restart
sudo ufw deny 111/tcp comment "Block rpcbind TCP"
sudo ufw deny 111/udp comment "Block rpcbind UDP"
sudo ufw deny 2049/tcp comment "Block NFS TCP"
sudo ufw deny 2049/udp comment "Block NFS UDP"
```

Rule: audit every listening port on every server. If you don't know why it's there, find out. If you don't need it, kill it.

Part 3: SSH Hardening

PermitRootLogin and PasswordAuthentication

Our load balancer was the oldest server in the fleet. When we checked its SSH config:

```
grep -E '(PermitRootLogin|PasswordAuthentication|PubkeyAuthentication)' /etc/ssh/sshd_config
```

The output showed defaults — which meant root login was allowed and password authentication was enabled. Default configurations are the enemy of security.

```
# /etc/ssh/sshd_config changes
# Make these changes, then restart SSH
```

```
# Disable root login entirely
PermitRootLogin no

# Disable password authentication - keys only
PasswordAuthentication no

# Explicitly enable key authentication
PubkeyAuthentication yes

# Disable empty passwords (belt and suspenders)
PermitEmptyPasswords no

# Limit to specific groups
AllowGroups ssh-users deploy-user

# Reduce grace time for authentication
LoginGraceTime 30

# Limit retry attempts
MaxAuthTries 3

# Disable X11 forwarding if not needed
X11Forwarding no

# Disable TCP forwarding if not needed for your use case
# (be careful - this may break tunnels you use)
# AllowTcpForwarding no
```

```
# Validate config before restarting
sudo sshd -t

# Restart SSH (keep your current session open!)
sudo systemctl restart sshd

# From a NEW terminal (don't close the old one), verify you can still connect
ssh -i your-key user@server
```

SSH Key Rotation When Onboarding New Machines

When we provisioned new servers, we had a bad habit: copy the `authorized_keys` from an existing server. Quick, easy, and a security nightmare. Old keys from departed engineers were accumulating.

```
#!/bin/bash
# rotate-ssh-keys.sh
# Run when onboarding new machines or auditing existing ones

SERVER=$1
```

```
AUTHORIZED_KEYS_FILE="/tmp/new_authorized_keys"

# Generate fresh authorized_keys from our source of truth (Vault or git)
vault kv get -field=authorized_keys secret/infrastructure/ssh-keys > "$AUTHORIZED_KEYS_FILE"

# Validate the file isn't empty
if [ ! -s "$AUTHORIZED_KEYS_FILE" ]; then
  echo "ERROR: authorized_keys file is empty. Aborting."
  exit 1
fi

# Count keys
KEY_COUNT=$(grep -c "^ssh-" "$AUTHORIZED_KEYS_FILE" || true)
echo "Deploying $KEY_COUNT SSH keys to $SERVER"

# Deploy via a temporary connection (you need access before rotation)
ssh-copy-id -f -i "$AUTHORIZED_KEYS_FILE" "deploy@$SERVER"

# Or use Ansible for fleet-wide rotation
ansible all -m authorized_key \
  -a "user=deploy key='{{ lookup(\"file\", \"$AUTHORIZED_KEYS_FILE\") }}' exclusive=yes" \
  --limit "$SERVER"

echo "Key rotation complete for $SERVER"
```

The `exclusive=yes` flag in Ansible is critical — it removes all keys not in your file. Without it, you're adding keys, not replacing them.

Service Restart After Password Rotation

We learned this lesson when we rotated the PostgreSQL password via Vault and the application kept working — for forty minutes, until the old connection pool connections closed. Then everything crashed.

Services hold open connections. Rotating a password doesn't terminate existing connections. You need to either:

1. Gracefully drain and restart the service, or
2. Use pg_terminate_backend to kill connections on the database side

```
#!/bin/bash
# restart-after-password-rotation.sh

SERVICE=$1
echo "Preparing to restart $SERVICE after credential rotation..."

case $SERVICE in
  "app-server")
    # Graceful restart - allow in-flight requests to complete
```

```
    sudo systemctl reload app-server || sudo systemctl restart app-server
    ;;
  "pgbouncer")
    # PgBouncer needs full restart to pick up new password
    sudo systemctl restart pgbouncer
    # Verify it connected successfully
    sleep 3
    psql -h localhost -p 6432 pgbouncer -c "SHOW POOLS;" || echo "WARNING: PgBouncer may not be
    ;;
  "all-app-servers")
    # Rolling restart across fleet
    for server in app1 app2 app3; do
      echo "Restarting $server..."
      ssh "$server" "sudo systemctl restart app-server"
      sleep 10  # Wait for health check to pass
    done
    ;;
esac

echo "Restart complete for $SERVICE"
```

Add password rotation to your runbook and include the restart step. Make it mandatory, not optional.

Part 4: Cleartext Credentials in Configuration Files

Patroni.yml — Passwords in Plain Sight

Patroni manages PostgreSQL high availability. Its configuration file includes database credentials. We found our `patroni.yml` with:

- World-readable permissions (`-rw-r--r-- root root`)
- Cleartext passwords for the replication user, the superuser, and the monitoring user
- Stored in a git repository "for backup purposes"

```
# DON'T DO THIS - patroni.yml before hardening
bootstrap:
  dcs:
    postgresql:
      use_pg_hba: true
  pg_hba:
    - host replication replicator 10.8.0.0/24 md5
postgresql:
  authentication:
    replication:
      username: replicator
      password: MyR3plicat1OnP@ss  # Cleartext in version control
```

```
    superuser:
      username: postgres
      password: Sup3rS3cr3tP@ss    # Cleartext in version control
    rewind:
      username: rewind
      password: R3w1ndP@ss         # Cleartext in version control
```

The fix has two parts: fix the file permissions, and use environment variable substitution.

```
# Immediate fix: restrict permissions
sudo chown postgres:postgres /etc/patroni/patroni.yml
sudo chmod 600 /etc/patroni/patroni.yml

# Verify
ls -la /etc/patroni/patroni.yml
# Should show: -rw------- postgres postgres
```

```
# patroni.yml after hardening - use environment variables
postgresql:
  authentication:
    replication:
      username: replicator
      password: "${PATRONI_REPLICATION_PASSWORD}"
    superuser:
      username: postgres
      password: "${PATRONI_SUPERUSER_PASSWORD}"
    rewind:
      username: rewind
      password: "${PATRONI_REWIND_PASSWORD}"
```

```
# /etc/patroni/patroni.env (chmod 600, owned by postgres)
PATRONI_REPLICATION_PASSWORD=MyR3plicat10nP@ss
PATRONI_SUPERUSER_PASSWORD=Sup3rS3cr3tP@ss
PATRONI_REWIND_PASSWORD=R3w1ndP@ss
```

```
# /etc/systemd/system/patroni.service
[Service]
User=postgres
EnvironmentFile=/etc/patroni/patroni.env
ExecStart=/usr/local/bin/patroni /etc/patroni/patroni.yml
```

Longer term, replace the `.env` file with Vault Agent or envconsul, which pulls secrets at runtime and injects them as environment variables without touching disk.

Hardcoded Credentials in Application Code

We found credentials hardcoded in a background job runner. The service connected to an external API — let's call it Antigravity — using credentials that had been "temporarily" hardcoded eight months ago and never moved to environment variables.

```python
# BEFORE - credentials hardcoded in source
class AntigravityClient:
    def __init__(self):
        self.api_key = "ag_prod_key_abc123xyz789"  # DO NOT COMMIT
        self.api_secret = "ag_secret_supersecretvalue"  # DO NOT COMMIT
        self.base_url = "https://api.antigravity.io/v2"
```

```python
# AFTER - credentials from environment
import os

class AntigravityClient:
    def __init__(self):
        self.api_key = os.environ["ANTIGRAVITY_API_KEY"]
        self.api_secret = os.environ["ANTIGRAVITY_API_SECRET"]
        self.base_url = os.environ.get(
            "ANTIGRAVITY_BASE_URL",
            "https://api.antigravity.io/v2"
        )

        if not self.api_key or not self.api_secret:
            raise ValueError(
                "ANTIGRAVITY_API_KEY and ANTIGRAVITY_API_SECRET must be set"
            )
```

After fixing the code, rotate the credentials immediately. Treat any credential that was ever in source code as compromised.

```bash
# Scan your codebase for common patterns
grep -rn "password\s*=\s*['\"]" --include="*.py" .
grep -rn "api_key\s*=\s*['\"]" --include="*.py" .
grep -rn "secret\s*=\s*['\"]" --include="*.py" .

# Better: use a dedicated secret scanner
pip install detect-secrets
detect-secrets scan . > .secrets.baseline
```

Part 5: Docker Security — The docker.sock Problem

Traefik and docker.sock

Traefik is a reverse proxy that automatically discovers Docker containers via the Docker socket. The classic setup mounts `/var/run/docker.sock` directly into the Traefik container.

```yaml
# INSECURE - Traefik with direct socket access
services:
  traefik:
    image: traefik:v3.0
```

```
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro  # DANGEROUS
```

Anyone who can execute code inside the Traefik container now has full Docker daemon access. They can list all containers, read environment variables (which contain your secrets), start new containers with `--privileged`, mount host filesystem paths — full container escape.

The fix is `docker-socket-proxy`, which sits between Traefik and the Docker socket and enforces a minimal API surface.

```
# docker-compose.yml - secure setup with docker-socket-proxy
services:
  # The proxy that mediates access to Docker socket
  docker-socket-proxy:
    image: tecnativa/docker-socket-proxy:latest
    restart: unless-stopped
    environment:
      # Allow only what Traefik needs
      CONTAINERS: 1       # Read container info
      SERVICES: 1         # Read service info (Swarm)
      TASKS: 1            # Read task info (Swarm)
      NETWORKS: 1         # Read network info
      EVENTS: 1           # Subscribe to events
      PING: 1             # Health check
      VERSION: 1          # API version negotiation
      # EXPLICITLY DENY dangerous operations
      POST: 0             # No container creation
      AUTH: 0             # No authentication changes
      SECRETS: 0          # No secret access
      BUILD: 0            # No image builds
      COMMIT: 0           # No container commits
      CONFIGS: 0          # No config access
      DISTRIBUTION: 0     # No registry operations
      EXEC: 0             # No exec into containers
      IMAGES: 0           # No image management
      INFO: 0             # No daemon info (contains sensitive data)
      SWARM: 0            # No Swarm management
      SYSTEM: 0           # No system operations
      VOLUMES: 0          # No volume management
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
    networks:
      - docker-proxy-net
    # ONLY accessible from internal network - never expose externally

  traefik:
    image: traefik:v3.0
    restart: unless-stopped
```

```
    depends_on:
      - docker-socket-proxy
    command:
      - "--providers.docker=true"
      - "--providers.docker.endpoint=tcp://docker-socket-proxy:2375"
      - "--providers.docker.exposedbydefault=false"
    networks:
      - docker-proxy-net
      - web
    ports:
      - "80:80"
      - "443:443"

networks:
  docker-proxy-net:
    internal: true  # No external access to this network
  web:
    external: true
```

Now Traefik can't execute commands inside containers, can't read environment variables from other containers, and can't create new containers. It can only read the routing labels it needs.

Part 6: HAProxy Stats — The Default Password Nobody Changed

HAProxy has a built-in statistics interface. Our configuration had:

```
stats auth admin:admin
```

The stats page exposed: number of active connections per backend, session rates, error rates, server health status, and backend server IP addresses. Not credentials directly, but enough operational intelligence to help an attacker understand your topology.

```
# haproxy.cfg - stats configuration after hardening

frontend stats
    bind *:8404
    # Only accessible from VPN, not public internet
    # (UFW handles this, but defense-in-depth)
    acl is_vpn_network src 10.8.0.0/24
    http-request deny unless is_vpn_network

    stats enable
    stats uri /haproxy-stats
    stats realm HAProxy\ Statistics
    stats auth haproxy_admin:$(HAPROXY_STATS_PASSWORD)
    stats refresh 10s
    stats hide-version  # Don't advertise HAProxy version
```

```
    stats show-legends
```

The password goes in Vault. You retrieve it at deploy time and substitute it into the config:

```
#!/bin/bash
# deploy-haproxy.sh

STATS_PASSWORD=$(vault kv get -field=stats_password secret/haproxy)

# Generate config with substituted password
envsubst < /etc/haproxy/haproxy.cfg.template > /etc/haproxy/haproxy.cfg
export HAPROXY_STATS_PASSWORD="$STATS_PASSWORD"
envsubst < /etc/haproxy/haproxy.cfg.template > /etc/haproxy/haproxy.cfg

# Validate config
haproxy -c -f /etc/haproxy/haproxy.cfg

# Reload (not restart - maintains existing connections)
sudo systemctl reload haproxy
```

Part 7: Vault Policies for Agent Access

The AI agent needs Vault access for legitimate operational tasks. But it shouldn't have access to everything. Here's the policy structure we use:

```
# vault-policy-agent-readonly.hcl
# For agents that only need to read configuration

path "secret/data/app/+/config" {
  capabilities = ["read"]
}

path "secret/data/infrastructure/+/endpoints" {
  capabilities = ["read"]
}

# Allow token self-lookup (so agent can check its own permissions)
path "auth/token/lookup-self" {
  capabilities = ["read"]
}

# Allow token renewal
path "auth/token/renew-self" {
  capabilities = ["update"]
}
```

```
# vault-policy-agent-ops.hcl
# For agents that perform operational tasks

# Read app configuration
path "secret/data/app/*" {
  capabilities = ["read"]
}

# Read infrastructure endpoints (not credentials)
path "secret/data/infrastructure/endpoints/*" {
  capabilities = ["read"]
}

# Write to a specific agent work area
path "secret/data/agent-workspace/*" {
  capabilities = ["create", "read", "update", "delete"]
}

# EXPLICITLY deny high-sensitivity paths
path "secret/data/infrastructure/database-credentials/*" {
  capabilities = ["deny"]
}

path "secret/data/vault-admin/*" {
  capabilities = ["deny"]
}

path "auth/*" {
  capabilities = ["deny"]
}

path "sys/*" {
  capabilities = ["deny"]
}
```

```
# Apply policies
vault policy write agent-readonly vault-policy-agent-readonly.hcl
vault policy write agent-ops vault-policy-agent-ops.hcl

# Create AppRoles for each agent type
vault write auth/approle/role/agent-readonly \
  token_policies="agent-readonly" \
  token_ttl=30m \
  token_max_ttl=2h

vault write auth/approle/role/agent-ops \
  token_policies="agent-ops" \
```

```
  token_ttl=1h \
  token_max_ttl=4h
```

Part 8: The AI Agent Security Checklist

After everything we've been through, here's the checklist we run before deploying any AI agent that touches infrastructure.

Pre-Deployment

```
[ ] Agent runs with dedicated service account (not root, not shared account)
[ ] Service account has minimal filesystem permissions
[ ] Vault AppRole created with least-privilege policy
[ ] Agent token TTL <= 2 hours
[ ] All secrets in Vault or environment variables - none in code or config
[ ] Docker socket access via docker-socket-proxy if needed
[ ] Network access restricted: agent can only reach services it needs
[ ] Conversation/log storage is encrypted at rest
[ ] Log retention policy defined (and short - logs containing context are sensitive)
```

Infrastructure Baseline

```
[ ] UFW enabled on ALL servers (verify: sudo ufw status)
[ ] PermitRootLogin no on ALL SSH configs
[ ] PasswordAuthentication no on ALL SSH configs
[ ] All database ports bound to VPN interface only
[ ] All metric exporter ports bound to VPN interface only
[ ] HAProxy stats using non-default credentials from Vault
[ ] Patroni.yml using environment variable substitution
[ ] patroni.yml permissions: 600, owned by postgres:postgres
[ ] No unused services running (NFS, rpcbind, etc.)
[ ] Traefik using docker-socket-proxy
```

Operational

```
[ ] Vault token rotation automated
[ ] SSH key rotation process documented and tested
[ ] Runbook includes service restart steps after credential rotation
[ ] Secret scanning running in CI/CD
[ ] Periodic port scan scheduled (weekly from external IP)
[ ] Incident response plan covers "AI agent exposed credentials" scenario
```

Lessons Learned

The agent is not security-aware. It will surface whatever information helps it complete the task. If debugging a Vault connection issue requires showing the token, it will show the token. You cannot train the AI to be more careful — you have to make the environment safe.

Default configurations are your enemy. Every service we found with a default password, default configuration, or no firewall had been set up quickly under time pressure. "I'll secure it properly later" is the most expensive statement in infrastructure.

Defense in depth is not optional. UFW rules AND network binding AND VPN-only routing. AppRole policies AND short TTLs AND token rotation. Multiple layers, because any single layer will eventually fail.

Audit everything, regularly. The server with UFW inactive had been that way for six months. Nobody checked because nobody had a scheduled process for checking. Add `ufw status`, `ss -tlnp`, and `sshd -T | grep -E '(PermitRoot|PasswordAuth)'` to your regular infrastructure audit scripts.

Rotate early, rotate often. When in doubt about whether a credential was exposed, rotate it. The cost of rotating a credential that wasn't actually exposed is an hour of work. The cost of not rotating one that was exposed is an incident, an audit, and potentially a breach notification.

The root token incident was embarrassing. But it forced us to fix six months of accumulated security debt in one week. The AI agent, in its naive helpfulness, did us a favor.

Next: Chapter 12 — AI Image Generation at Scale: what happens when you ask an AI to generate images for 289 blog posts and hit quota halfway through.

Chapter 12: AI Image Generation at Scale

> "We hit the quota limit at post 147 of 289. The batch had been running for four hours. I stared at the error message and thought: I should have tested this on ten posts first."

The 289-Post Problem

Every blog post needs a featured image. We had 289 blog posts going live as part of a content migration, and zero images. The designer quoted three weeks and $4,000. We had five days and a Vertex AI quota.

The pitch to leadership was straightforward: use Gemini to generate images in batch, review them, publish. Four hours of work, not three weeks.

What actually happened was two weeks of quota battles, script rewrites, delimiter disasters, and a hard lesson about the gap between "AI can generate images" and "AI can generate professional-quality images at scale."

This chapter covers the real production experience: the quota hits, the hot-swapping, the checkpointing, and the moment we realized AI-generated logos were a dead end.

The Architecture: Batch Processing at Scale

Starting Point: What We Had

We had a CSV of blog posts with titles, slugs, and meta descriptions. We needed to:

1. Generate a relevant image prompt from the post metadata
2. Call Vertex AI Imagen to generate the image
3. Save with the correct filename
4. Track progress so we could resume if anything failed

The initial script was 80 lines of Python and worked perfectly on the first ten posts.

```
#!/usr/bin/env python3
# generate-blog-images-batch.py
```

```
import csv
import os
import time
import json
import vertexai
from vertexai.vision_models import ImageGenerationModel
from pathlib import Path
from datetime import datetime

# Configuration
PROJECT_ID = os.environ["GOOGLE_CLOUD_PROJECT"]
LOCATION = "us-central1"
OUTPUT_DIR = Path("generated-images")
CHECKPOINT_FILE = Path("generation-checkpoint.json")
INPUT_CSV = "blog-posts.csv"

# Style prompt suffix - applied to every image
STYLE_SUFFIX = (
    "Professional blog hero image, clean modern design, "
    "tech-forward aesthetic, minimal text, high resolution, "
    "suitable for a DevOps and cloud computing blog"
)

def load_checkpoint() -> dict:
    """Load progress from previous run."""
    if CHECKPOINT_FILE.exists():
        with open(CHECKPOINT_FILE) as f:
            return json.load(f)
    return {"completed": [], "failed": [], "last_line": 0}

def save_checkpoint(checkpoint: dict):
    """Save progress - called after every successful generation."""
    with open(CHECKPOINT_FILE, "w") as f:
        json.dump(checkpoint, f, indent=2)

def build_image_prompt(title: str, description: str) -> str:
    """Build a Imagen prompt from post metadata."""
    return (
        f"Hero image for a blog post titled '{title}'. "
        f"Topic: {description[:200]}. "
        f"{STYLE_SUFFIX}"
    )

def generate_image(model, prompt: str, output_path: Path) -> bool:
    """Generate a single image. Returns True on success."""
    try:
        response = model.generate_images(
```

```
            prompt=prompt,
            number_of_images=1,
            aspect_ratio="16:9",
            safety_filter_level="block_some",
        )
        if response.images:
            response.images[0].save(str(output_path))
            return True
        return False
    except Exception as e:
        print(f"  ERROR: {e}")
        return False

def main():
    vertexai.init(project=PROJECT_ID, location=LOCATION)
    model = ImageGenerationModel.from_pretrained("imagegeneration@006")

    OUTPUT_DIR.mkdir(exist_ok=True)
    checkpoint = load_checkpoint()
    completed_slugs = set(checkpoint["completed"])

    print(f"Resuming from checkpoint: {len(completed_slugs)} posts already done")

    with open(INPUT_CSV, newline="", encoding="utf-8") as csvfile:
        reader = csv.DictReader(csvfile)
        rows = list(reader)

    total = len(rows)
    start_line = checkpoint.get("last_line", 0)

    for i, row in enumerate(rows[start_line:], start=start_line):
        slug = row["slug"]

        if slug in completed_slugs:
            continue

        output_path = OUTPUT_DIR / f"{slug}.png"
        title = row["title"]
        description = row.get("meta_description", row.get("excerpt", ""))

        print(f"[{i+1}/{total}] Generating: {title[:60]}...")

        prompt = build_image_prompt(title, description)
        success = generate_image(model, prompt, output_path)

        if success:
            checkpoint["completed"].append(slug)
```

```
            checkpoint["last_line"] = i + 1
            save_checkpoint(checkpoint)
            print(f"  Saved: {output_path}")
        else:
            checkpoint["failed"].append({"slug": slug, "line": i, "title": title})
            save_checkpoint(checkpoint)
            print(f"  FAILED: {slug}")

        # Rate limiting - Imagen has per-minute quotas
        time.sleep(2)

    print(f"\nComplete: {len(checkpoint['completed'])} generated, "
          f"{len(checkpoint['failed'])} failed")

if __name__ == "__main__":
    main()
```

This worked until post 147.

Hitting Quota Mid-Batch

The Error

```
google.api_core.exceptions.ResourceExhausted: 429 Quota exceeded for quota metric
'online_prediction_requests' and limit 'REQUESTS_PER_MINUTE_PER_PROJECT_PER_BASE_MODEL'
of service 'aiplatform.googleapis.com'.
```

Four hours of work. 147 images generated. 142 to go.

The default Vertex AI quota for image generation is conservative — enough for development and testing, not enough for a 289-post batch job.

Option 1: Request a Quota Increase (Takes Days)

```
# Via gcloud - check current quotas
gcloud compute project-info describe --project=$PROJECT_ID | grep -A 5 "REQUESTS"

# Quota increases via console: APIs & Services → Quotas
# Typical turnaround: 1-3 business days
```

We needed images in five days. Waiting for quota approval wasn't an option.

Option 2: Hot-Swap API Keys Across Projects

The fast path: create a second Google Cloud project, enable the Vertex AI API, and switch to it when the first project hits quota.

```python
# api-key-manager.py
# Manages multiple GCP project credentials with automatic rotation

import os
import time
import json
from dataclasses import dataclass, field
from typing import Optional
import vertexai
from vertexai.vision_models import ImageGenerationModel

@dataclass
class ProjectCredential:
    project_id: str
    credentials_file: str
    requests_this_minute: int = 0
    minute_start: float = field(default_factory=time.time)
    quota_exhausted_until: float = 0.0
    total_requests: int = 0

class RotatingProjectManager:
    """Manages multiple GCP projects with automatic failover on quota exhaustion."""

    def __init__(self, projects: list[dict]):
        self.projects = [ProjectCredential(**p) for p in projects]
        self.current_index = 0
        self.requests_per_minute_limit = 8  # Conservative - actual limit is 10

    def _reset_if_new_minute(self, project: ProjectCredential):
        now = time.time()
        if now - project.minute_start >= 60:
            project.requests_this_minute = 0
            project.minute_start = now

    def get_active_project(self) -> Optional[ProjectCredential]:
        """Get a project that has quota available."""
        now = time.time()
        # Try each project in rotation
        for _ in range(len(self.projects)):
            project = self.projects[self.current_index]
            self._reset_if_new_minute(project)

            # Skip if quota-exhausted cooldown is active
            if project.quota_exhausted_until > now:
                remaining = project.quota_exhausted_until - now
                print(f"  Project {project.project_id}: in cooldown ({remaining:.0f}s remaining
                self.current_index = (self.current_index + 1) % len(self.projects)
```

```
                continue

            # Skip if at per-minute limit
            if project.requests_this_minute >= self.requests_per_minute_limit:
                print(f"  Project {project.project_id}: at per-minute limit")
                self.current_index = (self.current_index + 1) % len(self.projects)
                continue

            return project

        return None  # All projects exhausted

    def mark_quota_exhausted(self, project: ProjectCredential, cooldown_seconds: int = 300):
        """Mark a project as quota-exhausted with a cooldown period."""
        project.quota_exhausted_until = time.time() + cooldown_seconds
        print(f"  Marked {project.project_id} as quota-exhausted for {cooldown_seconds}s")
        # Rotate to next project
        self.current_index = (self.current_index + 1) % len(self.projects)

    def record_request(self, project: ProjectCredential):
        project.requests_this_minute += 1
        project.total_requests += 1


def generate_with_rotation(manager: RotatingProjectManager, prompt: str, output_path) -> bool:
    """Generate an image, rotating projects on quota errors."""
    max_retries = len(manager.projects) * 2

    for attempt in range(max_retries):
        project = manager.get_active_project()

        if project is None:
            # All projects exhausted - wait for cooldowns
            wait_time = 60
            print(f"  All projects exhausted. Waiting {wait_time}s...")
            time.sleep(wait_time)
            continue

        try:
            # Switch to this project's credentials
            os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = project.credentials_file
            vertexai.init(project=project.project_id, location="us-central1")
            model = ImageGenerationModel.from_pretrained("imagegeneration@006")

            response = model.generate_images(
                prompt=prompt,
                number_of_images=1,
```

```
                aspect_ratio="16:9",
            )

            if response.images:
                response.images[0].save(str(output_path))
                manager.record_request(project)
                return True

        except Exception as e:
            error_str = str(e)
            if "ResourceExhausted" in error_str or "429" in error_str:
                manager.mark_quota_exhausted(project, cooldown_seconds=300)
            elif "403" in error_str or "credentials" in error_str.lower():
                print(f"  Credential error for {project.project_id}: {e}")
                manager.mark_quota_exhausted(project, cooldown_seconds=3600)
            else:
                print(f"  Unexpected error (attempt {attempt+1}): {e}")
                time.sleep(5)

    return False


# Usage
projects_config = [
    {
        "project_id": "my-project-prod",
        "credentials_file": "/secrets/gcp-prod-credentials.json"
    },
    {
        "project_id": "my-project-batch",
        "credentials_file": "/secrets/gcp-batch-credentials.json"
    },
    {
        "project_id": "my-project-overflow",
        "credentials_file": "/secrets/gcp-overflow-credentials.json"
    },
]

manager = RotatingProjectManager(projects_config)
```

With three projects, each with a quota of 10 requests per minute, we effectively had 30 requests per minute. The batch that had stalled at post 147 was running again within ten minutes.

The Delimiter Disaster

How the Bug Appeared

Blog post titles contain all kinds of characters. One of our posts was titled:

> "PostgreSQL vs MySQL vs SQLite | Which Database Should You Choose?"

Our CSV used | as a column delimiter.

```
# The CSV that caused problems
title|slug|meta_description
PostgreSQL vs MySQL vs SQLite | Which Database Should You Choose?|database-comparison|A compre
```

When the CSV reader hit that line, it saw five columns instead of three. The title field became `PostgreSQL vs MySQL vs SQLite`, the slug became `Which Database Should You Choose?`, and the meta description parser got confused.

The generated image for that post was titled something like "Hero image for 'PostgreSQL vs MySQL vs SQLite' " — truncating the title mid-comparison, dropping the actual question, and producing a generic database image instead of a comparison-focused one.

We only caught it during the review pass. 23 posts had malformed prompts.

The Fix: Auto-Detect and Switch to TAB Delimiter

```
#!/usr/bin/env python3
# csv-delimiter-detector.py

import csv
import sys
from pathlib import Path

def detect_and_fix_delimiter(input_path: str, output_path: str) -> str:
    """
    Detect if pipe delimiter causes column count inconsistency.
    If so, convert to TAB delimiter.
    Returns the delimiter used in the output file.
    """
    input_file = Path(input_path)

    # Read raw lines to check for pipe issues
    with open(input_file, encoding="utf-8") as f:
        lines = f.readlines()

    if not lines:
        raise ValueError("Empty input file")

    header = lines[0].strip()

    # Count expected columns from header
```

```
    pipe_count_header = header.count("|")
    tab_count_header = header.count("\t")

    # Determine which delimiter is more consistent
    if pipe_count_header > 0:
        # Check if any data rows have a different pipe count
        header_pipes = pipe_count_header
        inconsistent_rows = []
        for i, line in enumerate(lines[1:], start=2):
            if line.strip():
                row_pipes = line.strip().count("|")
                if row_pipes != header_pipes:
                    inconsistent_rows.append((i, line.strip()[:80]))

        if inconsistent_rows:
            print(f"WARNING: Found {len(inconsistent_rows)} rows with inconsistent pipe counts"
            print("Sample inconsistencies:")
            for line_num, sample in inconsistent_rows[:3]:
                print(f"  Line {line_num}: {sample}")
            print("Auto-converting to TAB delimiter...")
            return convert_to_tab(input_file, output_path)
        else:
            print("Pipe delimiter is consistent - no conversion needed")
            return "|"
    elif tab_count_header > 0:
        print("File already uses TAB delimiter")
        return "\t"
    else:
        print("Falling back to comma delimiter")
        return ","


def convert_to_tab(input_file: Path, output_path: str) -> str:
    """
    Convert a pipe-delimited CSV to TAB-delimited.
    Handles pipes within field values by quoting.
    """
    # First pass: read with pipe delimiter and proper quoting
    rows = []
    with open(input_file, encoding="utf-8", newline="") as f:
        # Use csv module with pipe delimiter
        reader = csv.reader(f, delimiter="|", quotechar='"')
        for row in reader:
            rows.append(row)

    if not rows:
        raise ValueError("No data after parsing")
```

```
    # Validate consistency
    expected_cols = len(rows[0])
    inconsistent = [(i+1, len(r)) for i, r in enumerate(rows) if len(r) != expected_cols]
    if inconsistent:
        print(f"WARNING: After pipe-parsing, {len(inconsistent)} rows have wrong column count")
        print("These rows may have unquoted pipe characters in values")
        # Attempt manual repair for common case
        rows = repair_pipe_rows(rows, expected_cols)

    # Write as TAB-delimited
    with open(output_path, "w", encoding="utf-8", newline="") as f:
        writer = csv.writer(f, delimiter="\t", quotechar='"', quoting=csv.QUOTE_MINIMAL)
        writer.writerows(rows)

    print(f"Converted {len(rows)} rows to TAB-delimited: {output_path}")
    return "\t"


def repair_pipe_rows(rows: list, expected_cols: int) -> list:
    """
    Attempt to repair rows with extra pipe characters.
    Strategy: merge excess columns, assuming last N fields are stable.
    """
    repaired = []
    for row in rows:
        if len(row) == expected_cols:
            repaired.append(row)
        elif len(row) > expected_cols:
            # Merge extra columns back into the title (first field)
            excess = len(row) - expected_cols
            merged_title = "|".join(row[:excess + 1])
            repaired_row = [merged_title] + row[excess + 1:]
            repaired.append(repaired_row)
            print(f"  Repaired: merged {excess+1} columns into title: {merged_title[:60]}")
        else:
            print(f"  WARNING: Row has too few columns ({len(row)} < {expected_cols}): {row}")
            repaired.append(row)
    return repaired


if __name__ == "__main__":
    input_csv = sys.argv[1] if len(sys.argv) > 1 else "blog-posts.csv"
    output_csv = sys.argv[2] if len(sys.argv) > 2 else "blog-posts-fixed.csv"

    delimiter = detect_and_fix_delimiter(input_csv, output_csv)
    print(f"Output file ready with delimiter: repr('{delimiter}')")
```

After this fix, we re-ran the 23 affected posts with correct prompts.

Checkpoint and Resume: Never Start Over

The Principle

Four hours of batch processing should never be thrown away. Every long-running batch job needs checkpointing. The design is simple:

- Write a checkpoint after every successful operation
- On restart, load the checkpoint and skip completed items
- Track failures separately — they may need manual review or retry

```
#!/usr/bin/env python3
# checkpoint-manager.py

import json
import time
from pathlib import Path
from dataclasses import dataclass, asdict
from typing import Optional

@dataclass
class GenerationCheckpoint:
    completed: list[str]        # Slugs successfully generated
    failed: list[dict]          # Failed items with error info
    skipped: list[str]          # Intentionally skipped
    last_processed_index: int   # Line number in input file
    session_start: str          # ISO timestamp of this session
    total_input_rows: int       # Total rows in input file
    api_requests_made: int      # Total API calls across all sessions

    @property
    def completion_percentage(self) -> float:
        if self.total_input_rows == 0:
            return 0.0
        return (len(self.completed) / self.total_input_rows) * 100

class CheckpointManager:
    def __init__(self, checkpoint_path: str = "generation-checkpoint.json"):
        self.path = Path(checkpoint_path)
        self._checkpoint: Optional[GenerationCheckpoint] = None

    def load_or_create(self, total_rows: int) -> GenerationCheckpoint:
        """Load existing checkpoint or create a fresh one."""
        if self.path.exists():
            with open(self.path) as f:
```

```
                data = json.load(f)
            self._checkpoint = GenerationCheckpoint(**data)
            print(f"Loaded checkpoint: {len(self._checkpoint.completed)}/{total_rows} complete
                  f"({self._checkpoint.completion_percentage:.1f}%)")
            print(f"Resuming from index {self._checkpoint.last_processed_index}")
        else:
            self._checkpoint = GenerationCheckpoint(
                completed=[],
                failed=[],
                skipped=[],
                last_processed_index=0,
                session_start=time.strftime("%Y-%m-%dT%H:%M:%SZ"),
                total_input_rows=total_rows,
                api_requests_made=0,
            )
            print(f"New checkpoint created for {total_rows} items")
        return self._checkpoint

    def save(self):
        """Save checkpoint to disk."""
        if self._checkpoint:
            # Atomic write - write to temp file then rename
            tmp_path = self.path.with_suffix(".tmp")
            with open(tmp_path, "w") as f:
                json.dump(asdict(self._checkpoint), f, indent=2)
            tmp_path.rename(self.path)

    def mark_complete(self, slug: str, index: int):
        if self._checkpoint:
            self._checkpoint.completed.append(slug)
            self._checkpoint.last_processed_index = index + 1
            self._checkpoint.api_requests_made += 1
            self.save()

    def mark_failed(self, slug: str, index: int, error: str, title: str = ""):
        if self._checkpoint:
            self._checkpoint.failed.append({
                "slug": slug,
                "index": index,
                "title": title,
                "error": error,
                "timestamp": time.strftime("%Y-%m-%dT%H:%M:%SZ")
            })
            self._checkpoint.last_processed_index = index + 1
            self.save()

    def is_complete(self, slug: str) -> bool:
```

```
        return self._checkpoint is not None and slug in self._checkpoint.completed

    def print_summary(self):
        if not self._checkpoint:
            return
        cp = self._checkpoint
        print(f"\n--- Checkpoint Summary ---")
        print(f"Completed:  {len(cp.completed)}/{cp.total_input_rows} "
              f"({cp.completion_percentage:.1f}%)")
        print(f"Failed:     {len(cp.failed)}")
        print(f"API calls:  {cp.api_requests_made}")
        if cp.failed:
            print(f"\nFailed items (review manually):")
            for item in cp.failed[:10]:
                print(f"  [{item['index']}] {item['slug']}: {item['error'][:80]}")


# Resume from a specific line number (for manual override)
def resume_from_line(checkpoint_path: str, line_number: int):
    """Manually override the resume point."""
    manager = CheckpointManager(checkpoint_path)
    if manager.path.exists():
        with open(manager.path) as f:
            data = json.load(f)
        data["last_processed_index"] = line_number
        with open(manager.path, "w") as f:
            json.dump(data, f, indent=2)
        print(f"Checkpoint updated: will resume from line {line_number}")
    else:
        print("No checkpoint file found")
```

Resuming After a Failure

```
# Normal resume - just run the script again
python3 generate-blog-images-batch.py

# Manual resume from a specific line (if checkpoint is corrupted)
python3 -c "
from checkpoint_manager import resume_from_line
resume_from_line('generation-checkpoint.json', 147)
"
python3 generate-blog-images-batch.py

# Check checkpoint status without running
python3 -c "
import json
with open('generation-checkpoint.json') as f:
```

```
    cp = json.load(f)
print(f'Completed: {len(cp[\"completed\"])}')
print(f'Failed: {len(cp[\"failed\"])}')
print(f'Resume from: {cp[\"last_processed_index\"]}')
"
```

The Logo Experiment: When AI Isn't the Right Tool

The Pitch

We needed a logo for the blog. We had Gemini. Why pay a designer?

The answer, after 47 logo generation attempts: because professional logo design requires vector graphics, brand strategy, and design system thinking that current AI image models don't provide.

Here's what we actually got:

Attempt 1-10: Generic tech logos. Blue circles, abstract circuits, the usual.

Attempt 11-20: Closer to our description, but inconsistent styling. Each image looked like it came from a different design agency.

Attempt 21-30: We started adding style references. Better aesthetics, but the text rendering was broken — letters merged, fonts were imaginary, words were gibberish.

Attempt 31-40: Focused on wordmark with specific font instructions. Gemini cannot reliably render specific fonts. The letter forms were wrong.

Attempt 41-47: Abstract marks only, no text. These were actually usable as brand elements, but not as logos — they were decorative.

```
# The prompts we tried (abbreviated)
logo_attempts = [
    # Phase 1 - too generic
    "Modern tech company logo, blue and white, professional",

    # Phase 2 - more specific
    "Minimalist logo for a DevOps blog called 'Infrastructure Atlas'. "
    "Single color, works at small sizes, no gradients, vector-style",

    # Phase 3 - style references
    "Logo in the style of Stripe or Linear: clean, modern, geometric. "
    "For 'Infrastructure Atlas' blog. Dark navy and electric blue",

    # Phase 4 - text focus
    "Wordmark logo: 'Infrastructure Atlas' in Inter font, bold weight, "
    "dark navy color, with a small abstract topographic map icon",

    # Phase 5 - give up on text
    "Abstract geometric mark only (no text): topographic contour lines "
```

```
    "forming a mountain shape, dark navy, suitable for logo use",
]
```

The verdict: AI image generation is excellent for blog hero images, social graphics, and illustrations. It is not a replacement for logo design, because:

1. Vector output is not available (raster only — doesn't scale)
2. Text rendering is unreliable for specific typography
3. Consistency across variations requires manual iteration that a designer does better
4. Brand strategy (color meaning, competitive differentiation) is out of scope

We hired a designer for the logo. $800, three revisions, delivered in four days. Worth it.

Rate Limits: The textembedding-gecko Bottleneck

The Situation

Alongside image generation, we were also building semantic search over our blog content. This required generating embeddings for each post using `text-embedding-gecko`.

```
# embedding-generator.py

import os
import time
from vertexai.language_models import TextEmbeddingModel

def generate_embeddings_with_rate_limit(texts: list[str], requests_per_minute: int = 4) -> lis
    """
    Generate embeddings with conservative rate limiting.
    Default: 4 RPM (below the 5 RPM quota limit).
    """
    model = TextEmbeddingModel.from_pretrained("textembedding-gecko@003")
    embeddings = []
    interval = 60.0 / requests_per_minute

    for i, text in enumerate(texts):
        start = time.time()

        try:
            result = model.get_embeddings([text])
            embeddings.append(result[0].values)
            print(f"  [{i+1}/{len(texts)}] Embedded ({len(result[0].values)} dims)")
        except Exception as e:
            if "ResourceExhausted" in str(e):
                print(f"  Rate limit hit at item {i+1}. Waiting 60s...")
                time.sleep(60)
                # Retry
                result = model.get_embeddings([text])
```

```
                embeddings.append(result[0].values)
            else:
                print(f"  Error at item {i+1}: {e}")
                embeddings.append(None)

        # Respect rate limit
        elapsed = time.time() - start
        sleep_time = max(0, interval - elapsed)
        if sleep_time > 0:
            time.sleep(sleep_time)

    return embeddings
```

The default quota was 5 requests per minute. For 289 posts, that was 57 minutes of embedding generation — assuming zero errors and perfect timing.

In practice: 90 minutes with error recovery.

Requesting a Quota Increase

The Google Cloud quota UI shows a slider for `textembedding-gecko`. The self-service maximum is 5 RPM — the same as the default. The slider doesn't go higher.

```
Current quota: 5 requests/minute
Maximum self-service quota: 5 requests/minute
```

To get above 5 RPM, you need to contact Google Cloud sales or support, explain your use case, and wait for a manual review.

```
# Check your current quota via gcloud
gcloud alpha services quota list \
  --service=aiplatform.googleapis.com \
  --consumer=project/$PROJECT_ID \
  --filter="metric:aiplatform.googleapis.com/online_prediction_requests"

# Create a support case for quota increase
# (No CLI for this - use console or email)
# support.google.com/cloud → Create Case → Quota Increase
```

While waiting for the quota increase (we requested 500 RPM, were granted 60 RPM after five days), we batched embedding requests:

```
def generate_embeddings_batched(texts: list[str], batch_size: int = 5) -> list:
    """
    Batch embedding requests - gecko supports up to 5 texts per request.
    One API call for 5 texts vs 5 API calls = 5x throughput.
    """
    model = TextEmbeddingModel.from_pretrained("textembedding-gecko@003")
    all_embeddings = []
```

```
    for i in range(0, len(texts), batch_size):
        batch = texts[i:i + batch_size]
        try:
            results = model.get_embeddings(batch)
            all_embeddings.extend([r.values for r in results])
            print(f"  Batch {i//batch_size + 1}: {len(batch)} texts embedded")
        except Exception as e:
            print(f"  Batch error: {e}")
            # Fall back to individual requests for this batch
            for text in batch:
                try:
                    result = model.get_embeddings([text])
                    all_embeddings.append(result[0].values)
                except Exception:
                    all_embeddings.append(None)
        time.sleep(12)  # 5 batches/min = 60s/5 = 12s between batches

    return all_embeddings
```

Batching reduced our embedding time from 90 minutes to 20 minutes — without any quota increase.

Veo 3: Video Generation

The Experiment

Halfway through the project, Veo 3 became available on Vertex AI. We decided to test it for short explainer clips to accompany select blog posts.

```
# veo3-video-generator.py (experimental)

import os
import time
import vertexai
from google.cloud import aiplatform

def generate_video_veo3(prompt: str, output_path: str, duration_seconds: int = 8) -> bool:
    """
    Generate a short video using Veo 3 via Vertex AI.
    Returns True on success.

    Note: Veo 3 is compute-intensive. Expect long generation times
    and occasional failures under high load.
    """
    client = aiplatform.gapic.PredictionServiceClient(
        client_options={"api_endpoint": "us-central1-aiplatform.googleapis.com"}
    )
```

```
    model_name = (
        f"projects/{os.environ['GOOGLE_CLOUD_PROJECT']}"
        f"/locations/us-central1"
        f"/publishers/google"
        f"/models/veo-3.0-generate-preview"
    )

    instance = {
        "prompt": prompt,
        "durationSeconds": duration_seconds,
        "aspectRatio": "16:9",
        "sampleCount": 1,
    }

    try:
        # Veo 3 is a long-running operation
        response = client.predict(
            endpoint=model_name,
            instances=[instance],
            timeout=300,  # 5 minutes - video generation is slow
        )

        if response.predictions:
            # Response contains base64-encoded video or a GCS URI
            prediction = response.predictions[0]
            # ... handle response format
            return True

    except Exception as e:
        error_str = str(e)
        if "503" in error_str or "high load" in error_str.lower():
            print(f"  Veo 3 high load failure (not your fault): {e}")
            return False
        elif "ResourceExhausted" in error_str:
            print(f"  Veo 3 quota exceeded: {e}")
            return False
        else:
            print(f"  Veo 3 error: {e}")
            return False

    return False
```

What Actually Happened

Authentication worked fine. The API calls went through. But under load, Veo 3 was returning 503 errors: "The server encountered a high load. Please try again."

We were not under high load. We were making one request at a time. The issue was that Veo 3 in preview was resource-constrained on Google's side, not ours.

```
[1/5] Generating intro video...
  Veo 3 high load failure: 503 The server is currently unable to handle the request
[1/5] Retry 1/3...
  Veo 3 high load failure: 503 The server is currently unable to handle the request
[1/5] Retry 2/3...
  Veo 3 high load failure: 503 The server is currently unable to handle the request
[1/5] All retries exhausted. Skipping video for: introduction-to-kubernetes
```

The lesson: preview models are not production-ready. We moved video generation to the backlog and shipped without it.

The POD Skill: Five Script Versions

The agent managing this workflow was using a skill called "POD" (Publish on Demand). Over the course of the project, the `add_text.py` and `generate_image_vertex.py` scripts inside the skill went through five versions:

- **v1:** Basic functionality, hardcoded credentials (caught immediately)
- **v2:** Environment variables, pipe delimiter, no error handling
- **v3:** TAB delimiter fix, basic retry logic, no checkpointing
- **v4:** Full checkpointing, quota rotation, batched embeddings
- **v5:** All of the above plus Veo 3 integration (partial, disabled in production)

The evolution of these scripts represents the actual learning curve of putting AI generation in production. You start simple and add complexity only when failure demands it.

```
# Skill directory structure after v5
~/.claude/skills/pod/
   add-text-to-image.py           # Text overlay on generated images
   generate-image-vertex.py       # Main image generation script
   checkpoint-manager.py          # Checkpoint/resume logic
   api-key-manager.py             # Multi-project quota rotation
   csv-delimiter-detector.py      # Delimiter fix utility
   embedding-generator.py         # textembedding-gecko wrapper
   config/
      projects.json               # GCP project credentials mapping
      style-prompts.json          # Reusable style prompt library
   README.md                      # Usage docs
```

Lessons Learned

Test on 10 before running on 289. The quota hit at 147 was avoidable. Run 10 posts, verify the output quality, check the quota consumption rate, and estimate completion time. Then scale.

Checkpointing is not optional for batch jobs. Any batch that takes more than 10 minutes needs a checkpoint. The cost of implementing it is one hour. The cost of not implementing it is potentially hours of re-work.

Delimiters matter. If your data contains the delimiter character (and user-generated content usually does), use a safe delimiter. TAB works for most content. If you need CSV compatibility, use proper quoting.

Know when AI isn't the right tool. Blog hero images: great. Social graphics: great. Logos: no. Text-heavy images with specific typography: no. Video in preview: not yet. The capability is real; the boundaries are also real.

Quota increases require lead time. If you're planning a large batch job, request quota increases a week before you need them. Self-service limits are conservative. Manual increases take days.

Batch API calls when possible. `textembedding-gecko` supports 5 texts per call. One call per text is 5x more expensive in quota terms than batching. Read the API documentation for batch limits before writing your generation loop.

The 289 posts shipped with AI-generated images. They look good — not indistinguishable from commissioned photography, but professional enough for a technical blog. Total cost: ~$120 in API fees, three developer-days of work, and the education covered in this chapter.

Next: Chapter 13 — Knowledge Management: building a RAG system with Cognee, Neo4j, and 400 DevOps books.

Chapter 13: Knowledge Management: RAG, Graphs, and Memory

> "We imported 400 DevOps books, 877MB of PDFs, into a graph-vector hybrid memory system. Then the agents started confusing Kubernetes docs with Wellington interior decorator SEO analysis. Turns out, dataset isolation is not automatic."

Why RAG Alone Isn't Enough

Every serious AI agent deployment eventually runs into the same wall: the agent doesn't remember what it did last week. You can stuff context into a prompt, but context windows fill up. You can use conversation history, but history gets expensive and eventually truncated. You can write notes to files, but files don't have semantic search.

Retrieval-Augmented Generation (RAG) solves the recall problem by giving agents a searchable external memory. But pure vector RAG — chunk text, embed, store, retrieve by cosine similarity — has its own limitations:

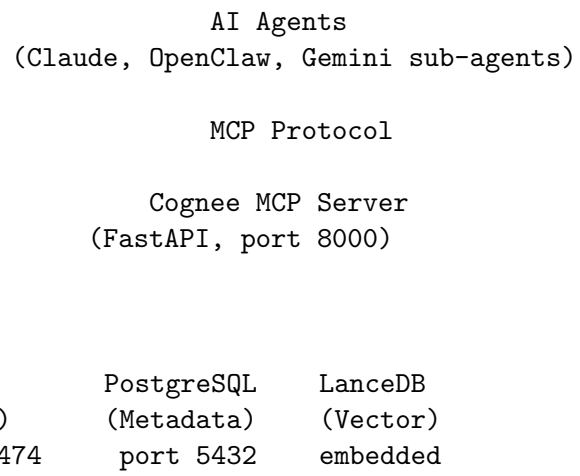
- It retrieves semantically similar chunks, not causally related ones
- It has no notion of "these two facts are related" unless they happen to be adjacent in the source text
- It can't answer "what did we decide last Tuesday" — it can only find documents that match "decision tuesday"
- It treats all knowledge as equally structured flat text

Graph-enhanced RAG adds a layer of structured relationships. You can ask: "What services depend on this database?" and get an answer built from traversing connections, not just matching keywords.

Cognee combines both approaches. This chapter covers what we actually built, the failures along the way, and what the system looks like running in production.

Architecture Overview

The Stack



Neo4j stores the knowledge graph: entities (concepts, services, technologies) and their relationships (depends-on, related-to, used-by, defined-in).

PostgreSQL stores document metadata, dataset memberships, import history, and the operational state of the Cognee pipeline.

LanceDB stores the vector embeddings for semantic similarity search. It runs embedded within the Cognee process — no separate service needed.

Cognee MCP Server exposes the memory system to AI agents via the Model Context Protocol. Agents call `cognify` to store knowledge and `search` to retrieve it.

Why Server-3?

We chose Server-3 for the full stack deployment. The decision factors:

- **RAM:** 9.3GB free after existing services — sufficient for Neo4j (2GB), PostgreSQL (1GB), and the Cognee process (1-2GB)
- **Disk:** 200GB available — needed for the 877MB book corpus plus index storage
- **Network:** Internal VPN routing only — no public exposure needed
- **Existing services:** No conflicting port usage on 7474, 7687, or 8000

```
# Pre-deployment resource check
free -h
# Mem: total 16G, used 6.7G, free 9.3G

df -h /
# /dev/sda1 500G 300G 200G 60% /

ss -tlnp | grep -E '(7474|7687|8000)'
# (no output - ports available)
```

Cognee Setup

Installation and Configuration

```
#!/bin/bash
# install-cognee-stack.sh

set -euo pipefail

# Create application directory
sudo mkdir -p /opt/cognee
sudo chown $USER:$USER /opt/cognee
cd /opt/cognee

# Python virtual environment
python3 -m venv .venv
source .venv/bin/activate

# Install Cognee with all backends
pip install cognee[neo4j,postgres,lancedb]

# Install MCP server dependencies
pip install "mcp[server]" fastapi uvicorn

# Verify installation
python3 -c "import cognee; print(cognee.__version__)"
```

```
# /opt/cognee/config.py
# Cognee configuration

import os
from cognee.api.v1.config import get_cognee_config

def configure_cognee():
    config = get_cognee_config()

    # Vector storage - LanceDB (embedded, no separate service)
    config.set("vector_db_provider", "lancedb")
    config.set("vector_db_path", "/opt/cognee/data/lancedb")

    # Graph storage - Neo4j
    config.set("graph_db_provider", "neo4j")
    config.set("graph_db_url", os.environ["NEO4J_URL"])          # bolt://10.8.0.12:7687
    config.set("graph_db_username", os.environ["NEO4J_USER"])
    config.set("graph_db_password", os.environ["NEO4J_PASSWORD"])
```

```
    # Relational storage - PostgreSQL
    config.set("db_provider", "postgres")
    config.set("db_host", os.environ["POSTGRES_HOST"])            # 10.8.0.12
    config.set("db_port", int(os.environ.get("POSTGRES_PORT", "5432")))
    config.set("db_name", os.environ["POSTGRES_DB"])              # cognee
    config.set("db_username", os.environ["POSTGRES_USER"])
    config.set("db_password", os.environ["POSTGRES_PASSWORD"])

    # LLM for knowledge extraction
    config.set("llm_provider", "anthropic")
    config.set("llm_model", os.environ.get("COGNEE_LLM_MODEL", "claude-haiku-4-5"))
    config.set("llm_api_key", os.environ["ANTHROPIC_API_KEY"])

    # Embedding model
    config.set("embedding_provider", "google")
    config.set("embedding_model", "textembedding-gecko@003")
    config.set("embedding_api_key", os.environ["GOOGLE_APPLICATION_CREDENTIALS"])

    return config
```

Docker Compose for Neo4j

```
# /opt/cognee/docker-compose.yml

services:
  neo4j:
    image: neo4j:5.18-community
    restart: unless-stopped
    environment:
      NEO4J_AUTH: "neo4j/${NEO4J_PASSWORD}"
      NEO4J_PLUGINS: '["apoc"]'
      NEO4J_dbms_memory_heap_initial__size: "512m"
      NEO4J_dbms_memory_heap_max__size: "2g"
      NEO4J_dbms_memory_pagecache_size: "512m"
    volumes:
      - neo4j-data:/data
      - neo4j-logs:/logs
    ports:
      # Bind to VPN interface only
      - "10.8.0.12:7474:7474"   # HTTP browser
      - "10.8.0.12:7687:7687"   # Bolt protocol
    networks:
      - cognee-internal

  cognee-api:
    build: .
```

```
    restart: unless-stopped
    environment:
      - NEO4J_URL=bolt://neo4j:7687
      - NEO4J_USER=neo4j
      - NEO4J_PASSWORD=${NEO4J_PASSWORD}
      - POSTGRES_HOST=10.8.0.12
      - POSTGRES_DB=cognee
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - ANTHROPIC_API_KEY=${ANTHROPIC_API_KEY}
      - GOOGLE_APPLICATION_CREDENTIALS=/secrets/gcp-credentials.json
    volumes:
      - /opt/cognee/data:/data
      - /secrets/gcp-credentials.json:/secrets/gcp-credentials.json:ro
    ports:
      - "10.8.0.12:8000:8000"
    depends_on:
      - neo4j
    networks:
      - cognee-internal

volumes:
  neo4j-data:
  neo4j-logs:

networks:
  cognee-internal:
    driver: bridge
```

Dataset Organization

The Four Datasets

We organized knowledge into four named datasets, each with a distinct purpose:

```
# dataset-config.py

DATASETS = {
    "seo-wellington-decorators": {
        "description": "SEO analysis for Wellington interior decorators client",
        "sources": ["keyword research", "competitor analysis", "content briefs"],
        "agents": ["seo-specialist"],
        "retention": "project-lifetime",
    },
    "openclaw-knowledge-devops-official": {
        "description": "Official documentation: Terraform, Kubernetes, AWS, GCP",
```

```
        "sources": ["docs.terraform.io", "kubernetes.io", "docs.aws.amazon.com"],
        "agents": ["openclaw", "claude-code"],
        "retention": "permanent",
        "update_frequency": "monthly",
    },
    "openclaw-knowledge-devops-community": {
        "description": "Community knowledge: blogs, war stories, real-world experience",
        "sources": ["medium", "dev.to", "personal blogs", "conference talks"],
        "agents": ["openclaw", "claude-code"],
        "retention": "permanent",
        "update_frequency": "weekly",
    },
    "openclaw-memory": {
        "description": "Long-term agent memory: decisions, outcomes, preferences",
        "sources": ["agent-generated", "conversation summaries"],
        "agents": ["openclaw", "claude-code"],
        "retention": "permanent",
        "update_frequency": "continuous",
    },
}
```

Why Separate Datasets Matter

We found out why dataset isolation matters the hard way. Before we implemented proper dataset boundaries, both agents were using the same default dataset.

The SEO analysis for the Wellington decorators client — keyword research, competitor analysis, content about curtain fabrics and paint colors — was mixed into the same vector space as Kubernetes documentation and Terraform modules.

When the infrastructure agent searched for "deployment strategies," it retrieved a mix of: - Kubernetes deployment strategy documentation (correct) - Blog posts about interior decorator business growth strategies (incorrect) - A content brief about decorating for different home styles (very incorrect)

The search results were diluted. Worse, the LLM sometimes incorporated the interior design context into responses about infrastructure, producing answers that were confidently wrong in subtle ways.

```
# cognee-dataset-manager.py
# Proper dataset isolation for multi-tenant agent setups

import cognee
from cognee.api.v1.cognify.cognify_router import cognify

async def add_to_dataset(
    content: str | list,
    dataset_name: str,
    metadata: dict = None
) -> dict:
```

```
    """
    Add content to a specific named dataset.
    Agents search within their assigned datasets only.
    """
    # Set active dataset context
    await cognee.prune.prune_system(metadata=True)  # Reset dataset context

    dataset = await cognee.datasets.get_or_create(name=dataset_name)

    if isinstance(content, str):
        content = [content]

    results = []
    for item in content:
        result = await cognee.add(
            data=item,
            dataset_name=dataset_name,
            metadata=metadata or {}
        )
        results.append(result)

    return {"dataset": dataset_name, "added": len(results)}


async def search_in_datasets(
    query: str,
    dataset_names: list[str],
    limit: int = 10
) -> list[dict]:
    """
    Search across specific datasets only.
    Never searches outside the specified dataset scope.
    """
    all_results = []

    for dataset_name in dataset_names:
        results = await cognee.search(
            query_text=query,
            query_type="SIMILARITY",
            datasets=[dataset_name],
            top_k=limit,
        )
        for r in results:
            r["source_dataset"] = dataset_name
            all_results.append(r)

    # Re-rank by score across all results
```

```
    all_results.sort(key=lambda x: x.get("score", 0), reverse=True)
    return all_results[:limit]
```

Importing 400 DevOps Books

The Import Pipeline

We had a collection of 838 PDF files, 877MB total — textbooks, O'Reilly books, conference proceedings, and reference guides covering Kubernetes, Terraform, AWS, observability, SRE practices, and security.

```
#!/usr/bin/env python3
# import-devops-books.py
# Import DevOps book collection into Cognee knowledge graph

import asyncio
import json
import os
import time
from pathlib import Path
from typing import Optional
import cognee

BOOKS_DIR = Path("/data/devops-books")
CHECKPOINT_FILE = Path("/opt/cognee/import-checkpoint.json")
TARGET_DATASET = "openclaw-knowledge-devops-official"
LOG_FILE = Path("/opt/cognee/import-log.jsonl")


def load_checkpoint() -> dict:
    if CHECKPOINT_FILE.exists():
        with open(CHECKPOINT_FILE) as f:
            return json.load(f)
    return {
        "completed": [],
        "failed": [],
        "last_index": 0,
        "total_files": 0,
        "session_start": time.strftime("%Y-%m-%dT%H:%M:%SZ"),
    }


def save_checkpoint(cp: dict):
    tmp = CHECKPOINT_FILE.with_suffix(".tmp")
    with open(tmp, "w") as f:
        json.dump(cp, f, indent=2)
```

```
    tmp.rename(CHECKPOINT_FILE)


def log_event(event: dict):
    with open(LOG_FILE, "a") as f:
        f.write(json.dumps({**event, "ts": time.strftime("%Y-%m-%dT%H:%M:%SZ")}) + "\n")


async def import_pdf(pdf_path: Path, dataset: str, retries: int = 3) -> bool:
    """Import a single PDF into Cognee. Returns True on success."""
    for attempt in range(retries):
        try:
            with open(pdf_path, "rb") as f:
                pdf_bytes = f.read()

            await cognee.add(
                data=pdf_bytes,
                dataset_name=dataset,
                metadata={
                    "filename": pdf_path.name,
                    "source_type": "pdf",
                    "import_session": "devops-books-batch-1",
                    "file_size_bytes": pdf_path.stat().st_size,
                }
            )
            return True

        except Exception as e:
            error_str = str(e)
            if "409" in error_str or "Conflict" in error_str:
                # Document already exists - treat as success
                log_event({"event": "already_exists", "file": pdf_path.name})
                return True
            elif "ResourceExhausted" in error_str or "429" in error_str:
                wait = 60 * (attempt + 1)
                print(f"  Rate limit. Waiting {wait}s (attempt {attempt+1}/{retries})")
                await asyncio.sleep(wait)
            else:
                print(f"  Error (attempt {attempt+1}/{retries}): {e}")
                if attempt < retries - 1:
                    await asyncio.sleep(10)

    return False


async def main():
    pdf_files = sorted(BOOKS_DIR.glob("**/*.pdf"))
```

```
    total = len(pdf_files)
    print(f"Found {total} PDF files ({sum(f.stat().st_size for f in pdf_files) / 1e6:.0f}MB)")

    cp = load_checkpoint()
    cp["total_files"] = total
    completed_set = set(cp["completed"])

    print(f"Resuming: {len(completed_set)}/{total} already imported")

    for i, pdf_path in enumerate(pdf_files[cp["last_index"]:], start=cp["last_index"]):
        rel_path = str(pdf_path.relative_to(BOOKS_DIR))

        if rel_path in completed_set:
            continue

        print(f"[{i+1}/{total}] {rel_path} ({pdf_path.stat().st_size / 1e6:.1f}MB)")

        success = await import_pdf(pdf_path, TARGET_DATASET)

        if success:
            cp["completed"].append(rel_path)
            log_event({"event": "imported", "file": rel_path, "index": i})
        else:
            cp["failed"].append({"file": rel_path, "index": i})
            log_event({"event": "failed", "file": rel_path, "index": i})

        cp["last_index"] = i + 1
        save_checkpoint(cp)

        # Throttle to avoid overwhelming the pipeline
        await asyncio.sleep(3)

    print(f"\nImport complete: {len(cp['completed'])} succeeded, {len(cp['failed'])} failed")
    if cp["failed"]:
        print("Failed files:")
        for item in cp["failed"]:
            print(f"  {item['file']}")


if __name__ == "__main__":
    asyncio.run(main())
```

War Story: The 409 Conflict Storm

About 200 files into the import, we started seeing a cascade of 409 Conflict errors.

```
[203/838] kubernetes-in-action-2nd-edition.pdf
  Error (attempt 1/3): 409 Conflict - Document 'kubernetes-in-action-2nd-edition.pdf'
```

```
already exists in dataset
Error (attempt 2/3): 409 Conflict
Error (attempt 3/3): 409 Conflict
FAILED: kubernetes-in-action-2nd-edition.pdf
```

The script was retrying 409 errors as if they were transient failures. A 409 Conflict means the document already exists — retrying will always produce the same error.

The fix was immediate: treat 409 as success (idempotent import).

```
# The fix - in import_pdf()
if "409" in error_str or "Conflict" in error_str:
    # Document already exists - this is fine
    # Cognee uses content hashing to deduplicate
    log_event({"event": "already_exists", "file": pdf_path.name})
    return True  # Not a failure
```

After the fix, we realized these 409s were from a previous partial import attempt. The checkpoint from that run had been lost when the server rebooted. The idempotent handling meant re-running was safe.

Lesson: Design import pipelines to be idempotent from the start. "Already exists" is not an error — it's confirmation.

War Story: Vertex AI Rate Limits Stall Everything

Mid-import, everything stalled. Not crashed — stalled. The script was running but progress had slowed to one file every 3-4 minutes.

```
# Diagnose what's happening
tail -f /opt/cognee/import-log.jsonl | grep -E '(rate_limit|429|ResourceExhausted)'
```

The Cognee cognify pipeline — which extracts entities and relationships from text — was calling the text embedding API for every chunk of every document. With 838 files, each averaging 100+ pages, we were generating thousands of embedding requests.

We had hit the `textembedding-gecko` quota. The same 5 RPM limit from Chapter 12.

Cognee doesn't expose direct rate limit controls in its public API. The embedding calls happen inside the cognify pipeline. Our options:

1. Wait for quota increase (days)
2. Switch embedding model to one with higher quota
3. Throttle the import pipeline (artificially slow it down to stay under quota)

We went with option 3 first, then option 2.

```
# Throttled import - add delay between files
# Calculate: 5 RPM, average doc generates ~20 embedding calls
# 20 calls / 5 RPM = 4 minutes per document minimum
# Add buffer: sleep 5 minutes between large documents

async def import_with_throttle(pdf_path: Path, dataset: str) -> bool:
```

```
    size_mb = pdf_path.stat().st_size / 1e6

    success = await import_pdf(pdf_path, dataset)

    # Throttle based on file size (larger = more chunks = more embedding calls)
    if size_mb > 10:
        wait = 300  # 5 min for large files
    elif size_mb > 5:
        wait = 120  # 2 min for medium files
    else:
        wait = 30   # 30s for small files

    print(f"  Throttling {wait}s (file size: {size_mb:.1f}MB)")
    await asyncio.sleep(wait)

    return success
```

The import completed over four days instead of four hours. Not ideal, but it completed.

The MCP Server: Agent-Accessible Knowledge

MCP Configuration

The MCP (Model Context Protocol) server exposes Cognee's search and storage capabilities to AI agents as callable tools.

```
# /opt/cognee/mcp-server.py
# MCP server exposing Cognee as agent tools

import asyncio
import json
from typing import Any
import cognee
from mcp.server import Server
from mcp.server.models import InitializationOptions
from mcp.types import (
    CallToolRequest,
    CallToolResult,
    ListToolsResult,
    TextContent,
    Tool,
)
import mcp.server.stdio

server = Server("cognee-memory")

@server.list_tools()
```

```
async def handle_list_tools() -> ListToolsResult:
    return ListToolsResult(
        tools=[
            Tool(
                name="cognify",
                description=(
                    "Store information in long-term agent memory. "
                    "Use for decisions, findings, configurations, and facts "
                    "that should persist across sessions."
                ),
                inputSchema={
                    "type": "object",
                    "properties": {
                        "content": {
                            "type": "string",
                            "description": "The information to store"
                        },
                        "dataset": {
                            "type": "string",
                            "description": "Dataset name (e.g., 'openclaw-memory')",
                            "default": "openclaw-memory"
                        },
                        "tags": {
                            "type": "array",
                            "items": {"type": "string"},
                            "description": "Optional tags for categorization"
                        }
                    },
                    "required": ["content"]
                }
            ),
            Tool(
                name="search_memory",
                description=(
                    "Search agent memory for relevant information. "
                    "Returns semantically similar content and related graph nodes."
                ),
                inputSchema={
                    "type": "object",
                    "properties": {
                        "query": {
                            "type": "string",
                            "description": "What to search for"
                        },
                        "datasets": {
                            "type": "array",
                            "items": {"type": "string"},
```

```
                            "description": "Datasets to search (omit for default agent datasets
                        },
                        "limit": {
                            "type": "integer",
                            "default": 10,
                            "description": "Max results to return"
                        }
                    },
                    "required": ["query"]
                }
            ),
            Tool(
                name="get_graph_neighbors",
                description=(
                    "Get related concepts from the knowledge graph. "
                    "Use to explore relationships between entities."
                ),
                inputSchema={
                    "type": "object",
                    "properties": {
                        "entity": {
                            "type": "string",
                            "description": "Entity name to explore"
                        },
                        "depth": {
                            "type": "integer",
                            "default": 2,
                            "description": "Graph traversal depth"
                        }
                    },
                    "required": ["entity"]
                }
            ),
        ]
    )


@server.call_tool()
async def handle_call_tool(request: CallToolRequest) -> CallToolResult:
    try:
        if request.name == "cognify":
            content = request.arguments.get("content", "")
            dataset = request.arguments.get("dataset", "openclaw-memory")
            tags = request.arguments.get("tags", [])

            await cognee.add(
                data=content,
```

```
            dataset_name=dataset,
            metadata={"tags": tags}
        )
        await cognee.cognify(datasets=[dataset])

        return CallToolResult(
            content=[TextContent(
                type="text",
                text=json.dumps({
                    "status": "stored",
                    "dataset": dataset,
                    "content_preview": content[:100] + "..." if len(content) > 100 else co
                })
            )]
        )

    elif request.name == "search_memory":
        query = request.arguments.get("query", "")
        datasets = request.arguments.get("datasets", ["openclaw-memory"])
        limit = request.arguments.get("limit", 10)

        results = await cognee.search(
            query_text=query,
            query_type="SIMILARITY",
            datasets=datasets,
            top_k=limit,
        )

        formatted = [
            {
                "content": r.get("text", r.get("content", ""))[:500],
                "score": round(r.get("score", 0), 4),
                "source": r.get("source_dataset", "unknown"),
                "metadata": r.get("metadata", {}),
            }
            for r in results
        ]

        return CallToolResult(
            content=[TextContent(
                type="text",
                text=json.dumps({"results": formatted, "count": len(formatted)})
            )]
        )

    elif request.name == "get_graph_neighbors":
        entity = request.arguments.get("entity", "")
```

```
            depth = request.arguments.get("depth", 2)

            # Query Neo4j directly for graph traversal
            results = await cognee.search(
                query_text=entity,
                query_type="GRAPH_COMPLETION",
                top_k=20,
            )

            return CallToolResult(
                content=[TextContent(
                    type="text",
                    text=json.dumps({"entity": entity, "neighbors": results})
                )]
            )

    except Exception as e:
        return CallToolResult(
            content=[TextContent(
                type="text",
                text=json.dumps({"error": str(e), "tool": request.name})
            )],
            isError=True
        )


async def main():
    async with mcp.server.stdio.stdio_server() as (read_stream, write_stream):
        await server.run(
            read_stream,
            write_stream,
            InitializationOptions(
                server_name="cognee-memory",
                server_version="1.0.0",
            ),
        )


if __name__ == "__main__":
    asyncio.run(main())
```

Claude Code MCP Configuration

```
// ~/.claude/mcp-config.json
{
  "mcpServers": {
    "cognee-memory": {
```

```
      "command": "/opt/cognee/.venv/bin/python3",
      "args": ["/opt/cognee/mcp-server.py"],
      "env": {
        "NEO4J_URL": "bolt://10.8.0.12:7687",
        "NEO4J_USER": "neo4j",
        "NEO4J_PASSWORD": "${NEO4J_PASSWORD}",
        "POSTGRES_HOST": "10.8.0.12",
        "POSTGRES_DB": "cognee",
        "POSTGRES_USER": "${POSTGRES_USER}",
        "POSTGRES_PASSWORD": "${POSTGRES_PASSWORD}",
        "ANTHROPIC_API_KEY": "${ANTHROPIC_API_KEY}"
      }
    }
  }
}
```

Crawl4AI vs Firecrawl: Choosing Your Web Ingestion Tool

We evaluated both tools for populating the community knowledge dataset with blog posts, documentation, and forum threads.

Firecrawl

Firecrawl is a full-site crawling service. You give it a domain and it follows links, respects robots.txt, handles JavaScript rendering, and returns clean markdown.

Strengths: - Full-site crawl with sitemap support - Deep crawl (follow all links to configured depth) - Handles JavaScript-heavy sites - Returns structured markdown, not raw HTML

Weaknesses: - Heavy infrastructure: requires Node.js, Redis, and the Firecrawl service itself - Self-hosted deployment is non-trivial - More expensive in API calls for targeted single-page extraction

Deployment:

```
# firecrawl-docker-compose.yml
services:
  firecrawl-api:
    image: mendableai/firecrawl:latest
    environment:
      - REDIS_URL=redis://redis:6379
      - PLAYWRIGHT_MICROSERVICE_URL=http://playwright:3000
    ports:
      - "3002:3002"
    depends_on:
      - redis
      - playwright
```

```
  redis:
    image: redis:7-alpine
    volumes:
      - redis-data:/data

  playwright:
    image: mendableai/firecrawl-playwright-service:latest
    ports:
      - "3000:3000"

volumes:
  redis-data:
```

```python
# firecrawl-ingestion.py

import requests
import time

FIRECRAWL_URL = "http://10.8.0.12:3002"

def crawl_site(url: str, max_depth: int = 3, limit: int = 100) -> list[dict]:
    """Crawl an entire site and return clean markdown for each page."""
    response = requests.post(
        f"{FIRECRAWL_URL}/v1/crawl",
        json={
            "url": url,
            "maxDepth": max_depth,
            "limit": limit,
            "scrapeOptions": {
                "formats": ["markdown"],
                "excludePatterns": ["*/tag/*", "*/category/*", "*/author/*"],
            }
        },
        timeout=30,
    )
    response.raise_for_status()

    job_id = response.json()["id"]
    print(f"Crawl job started: {job_id}")

    # Poll for completion
    while True:
        status_response = requests.get(f"{FIRECRAWL_URL}/v1/crawl/{job_id}")
        status_data = status_response.json()

        if status_data["status"] == "completed":
            return status_data.get("data", [])
        elif status_data["status"] == "failed":
```

```
            raise Exception(f"Crawl failed: {status_data.get('error')}")

        print(f"  Crawling... {status_data.get('completed', 0)}/{status_data.get('total', '?')}
        time.sleep(5)
```

Crawl4AI

Crawl4AI is a lightweight Python library for single-page extraction with structured output schemas.

Strengths: - Pure Python — no Node.js, no Redis, minimal dependencies - Excellent for targeted extraction with schemas - Fast for single-page or small-batch use cases - Configurable extraction schemas for structured data

Weaknesses: - No built-in full-site crawling (you manage link following yourself) - Less sophisticated JavaScript rendering than Firecrawl

```
# crawl4ai-ingestion.py

import asyncio
from crawl4ai import AsyncWebCrawler, BrowserConfig, CrawlerRunConfig
from crawl4ai.extraction_strategy import JsonCssExtractionStrategy
import json

# Schema for extracting blog post structure
BLOG_POST_SCHEMA = {
    "name": "BlogPost",
    "baseSelector": "article",
    "fields": [
        {"name": "title", "selector": "h1", "type": "text"},
        {"name": "author", "selector": ".author-name", "type": "text"},
        {"name": "date", "selector": "time", "type": "attribute", "attribute": "datetime"},
        {"name": "content", "selector": ".post-content", "type": "text"},
        {"name": "tags", "selector": ".tag", "type": "list", "fields": [
            {"name": "tag", "type": "text"}
        ]},
    ]
}

async def extract_blog_post(url: str) -> dict | None:
    """Extract structured content from a single blog post."""
    browser_config = BrowserConfig(headless=True, verbose=False)
    run_config = CrawlerRunConfig(
        extraction_strategy=JsonCssExtractionStrategy(BLOG_POST_SCHEMA),
        cache_mode="enabled",  # Cache responses for reruns
    )

    async with AsyncWebCrawler(config=browser_config) as crawler:
        result = await crawler.arun(url=url, config=run_config)
```

```
        if result.success and result.extracted_content:
            try:
                data = json.loads(result.extracted_content)
                if data:
                    return data[0]  # First matching article
            except json.JSONDecodeError:
                # Fall back to markdown content
                return {"content": result.markdown, "url": url}

    return None


async def batch_extract_posts(urls: list[str], concurrency: int = 3) -> list[dict]:
    """Extract multiple blog posts with controlled concurrency."""
    semaphore = asyncio.Semaphore(concurrency)
    results = []

    async def extract_with_limit(url: str):
        async with semaphore:
            result = await extract_blog_post(url)
            if result:
                result["source_url"] = url
                results.append(result)
            await asyncio.sleep(1)  # Polite delay

    await asyncio.gather(*[extract_with_limit(url) for url in urls])
    return results
```

Decision Matrix

Factor	Firecrawl	Crawl4AI
Full-site crawl	Yes	Manual
Infrastructure overhead	High (Node, Redis)	Minimal
JavaScript rendering	Excellent	Good
Structured extraction	Basic	Excellent (schemas)
Setup time	2-4 hours	15 minutes
Best for	Entire sites	Targeted pages

Our decision: Firecrawl for official documentation sites (crawl everything), Crawl4AI for targeted extraction of specific blog posts and articles. They complement each other.

Neural Memory: What the Graph Looks Like

After completing the full import — 400 books, Firecrawl runs across documentation sites, and targeted Crawl4AI extractions — we ran `cognee.get_graph_statistics()` to see what we'd built.

```python
# check-graph-stats.py

import asyncio
import cognee

async def main():
    stats = await cognee.get_graph_statistics()
    print(f"Nodes (neurons):  {stats['node_count']:,}")
    print(f"Edges (synapses): {stats['edge_count']:,}")
    print(f"Node types: {stats['node_types']}")
    print(f"Edge types: {stats['edge_types']}")

    # Sample some random nodes
    sample = await cognee.search(
        query_text="Kubernetes deployment",
        query_type="GRAPH_COMPLETION",
        top_k=5
    )
    for node in sample:
        print(f"  - {node.get('text', '')[:80]}")

asyncio.run(main())
```

Output after full setup:

```
Nodes (neurons):  6,839
Edges (synapses): 78,431
Node types: ['Concept', 'Technology', 'Person', 'Organization', 'Document', 'Procedure']
Edge types: ['RELATES_TO', 'DEPENDS_ON', 'DEFINED_IN', 'USED_BY', 'REFERENCES', 'SUPERSEDES']
```

6,839 nodes and 78,431 edges. The graph represents the conceptual structure of our operational knowledge — concepts connected to the documents that define them, technologies connected to the procedures that use them, procedures connected to the failures that motivated them.

When an agent searches for "Kubernetes pod scheduling," it gets back not just similar text, but related concepts: node affinity, resource requests, pod disruption budgets, and the specific pages in our imported books that cover each. Graph traversal turns a keyword search into a knowledge navigation session.

Lessons Learned

Dataset isolation is architecture, not configuration. Putting all your knowledge in one namespace is equivalent to putting all your code in one file. You will eventually regret it. Design your dataset structure before you start importing.

Idempotent imports from day one. Treat "already exists" as success. Network failures, reboots, and quota exhaustion will interrupt long import jobs. Your pipeline needs to resume from where it stopped without re-processing completed items.

Rate limits are the ceiling, not the target. The textembedding-gecko 5 RPM limit isn't something to operate at — it's something to operate well below, with room for retries. Build throttling that keeps you at 80% of the limit.

Graph memory and vector memory serve different queries. Vector: "find me something similar to this." Graph: "what else is connected to this concept?" You need both. Pure vector RAG answers questions about content. Graph-enhanced RAG answers questions about relationships.

Crawl4AI and Firecrawl are complementary. Choose based on the source. High-quality documentation sites with deep structure: Firecrawl. Targeted extraction with known schemas: Crawl4AI. Don't pick one and force it for everything.

Monitor the pipeline, not just the output. We only discovered the rate limit stall by watching log output. Add metrics: embedding calls per minute, documents in queue, processing rate, error frequency. A stalled pipeline that doesn't error is worse than one that fails loudly.

The system we built isn't perfect — the dataset isolation story is a cautionary tale about skipping architectural decisions under time pressure. But 6,839 nodes and 78,431 synapses later, the agents have access to a decade of collective DevOps knowledge, retrievable in milliseconds.

Next: Chapter 14 — Multi-Agent Orchestration in Practice: tmux, shared inboxes, and what happens when a sub-agent can't find a folder that doesn't exist.

Chapter 14: Multi-Agent Orchestration in Practice

> "The sub-agent reported success. The deployment had failed. The folder it was supposed to create files in didn't exist. Nobody checked. The agent just… moved on."

The Problem with One Agent

A single AI agent has a context window. Once that window fills, older context gets dropped. On a complex infrastructure task — say, migrating a PostgreSQL cluster while updating application configs and rotating credentials — you will fill that window before you finish.

The naive solution is to ask the agent to be more concise, to summarize as it goes, to be efficient. This works for a while. Then you hit 200K tokens on a multi-hour task, and the agent starts forgetting what it decided two hours ago.

The real solution is orchestration: break the work into scoped tasks, assign each task to a fresh agent with a full context window, and coordinate the results. This is multi-agent orchestration, and it's messier in practice than it sounds in theory.

This chapter covers what we actually built: the OpenClaw gateway architecture, the tmux-based communication channel, the failure modes we didn't anticipate, and the operational patterns that emerged from running this in production.

Architecture: OpenClaw Gateway Pattern

The Core Design

OpenClaw is our primary orchestrating agent — always running, always reachable. It owns the task queue, spawns sub-agents for heavy work, and coordinates results. Claude Code is a peer agent with complementary capabilities: deeper code analysis, file system operations, and direct access to the development environment.

```
User / Operator
```

```
                  OpenClaw Gateway
          (Primary orchestrating agent)
  - Task routing          - Provider fallback
  - Context management    - Sub-agent lifecycle
  - Cross-server comms    - Model policy enforcement



  Sub-agents                    Claude Code
  (Claude/Gemini)               (Peer agent)
  - Scoped tasks                - Dev tasks
  - Fresh context               - Code review
  - Auto-cleanup                - File ops
```

Shared Workspace Communication

OpenClaw and Claude Code communicate through a shared workspace — a directory both agents can read and write. This is a simple, reliable message-passing mechanism that doesn't require any real-time protocol.

```
~/.openclaw/workspace/
  inbox/
      from-openclaw.md        # Messages TO Claude Code
      from-claudecode.md      # Messages FROM Claude Code
  proposals/                   # Collaboration proposals
  research/                    # Research results
  MEMORY.md                    # Shared persistent memory
  memory/
      YYYY-MM-DD.md            # Daily operational logs
```

```
# OpenClaw sends a task to Claude Code
cat >> ~/.openclaw/workspace/inbox/from-openclaw.md << 'EOF'

## Task: Review PostgreSQL Migration Script
Timestamp: 2026-03-01T14:32:00Z
Priority: HIGH

Please review /opt/migrations/pg-migration-v2.sql for:
1. Data integrity risks
2. Lock contention issues
3. Rollback safety

Reply in from-claudecode.md when complete.
EOF

# Notify Claude Code via system event
```

```
openclaw system event --text "Task waiting in inbox/from-openclaw.md - PostgreSQL migration rev
```

```
# Claude Code checks inbox (should happen at session start and after idle)
cat ~/.openclaw/workspace/inbox/from-openclaw.md 2>/dev/null

# Claude Code replies after completing the review
cat >> ~/.openclaw/workspace/inbox/from-claudecode.md << 'EOF'

## Re: Review PostgreSQL Migration Script
Timestamp: 2026-03-01T14:51:00Z
Status: COMPLETE

Review findings:
1. Line 47: ALTER TABLE with full table rewrite on 50M row table - expect 20-40min lock
2. No explicit transaction boundaries - if script fails midway, partial state
3. Missing ROLLBACK statements for FK constraint changes

Recommended: Add explicit BEGIN/COMMIT, add LOCK TABLE with NOWAIT check first.
Detailed review: /tmp/pg-migration-review.md
EOF

openclaw system event --text "Migration review complete. 3 issues found. Details in inbox/from-
```

Tmux-Based Communication: Sending Keystrokes Between Sessions

The shared workspace approach works for asynchronous messages. But sometimes OpenClaw needs to inject a task into an active Claude Code session immediately — not "check the inbox when you get a chance," but "here's a task, start now."

This is where tmux comes in.

How ping-claudecode.sh Works

```
#!/bin/bash
# ping-claudecode.sh
# Injects a message into the active Claude Code tmux session

set -euo pipefail

MESSAGE="${1:-Check inbox for new task}"
SESSION_NAME="${CLAUDE_TMUX_SESSION:-claude-code}"
PANE_TARGET="${SESSION_NAME}:0.0"

# Verify the session exists
```

```
if ! tmux has-session -t "$SESSION_NAME" 2>/dev/null; then
    echo "ERROR: tmux session '$SESSION_NAME' not found"
    echo "Active sessions: $(tmux list-sessions 2>/dev/null || echo 'none')"
    exit 1
fi

# Write the message to the inbox first
INBOX_FILE="$HOME/.openclaw/workspace/inbox/from-openclaw.md"
cat >> "$INBOX_FILE" << EOF

## Incoming Task [$(date -u +%Y-%m-%dT%H:%M:%SZ)]
$MESSAGE
EOF

# Inject the message as keystrokes into the Claude Code pane
# This triggers Claude Code to process the message immediately
tmux send-keys -t "$PANE_TARGET" "" Enter
sleep 0.3
tmux send-keys -t "$PANE_TARGET" "New task from OpenClaw: $MESSAGE" Enter

echo "Message sent to $PANE_TARGET"
```

```
# Usage from OpenClaw
ping-claudecode.sh "Deploy the reviewed migration script to staging - approval granted"

# Or from shell directly
~/.openclaw/scripts/ping-claudecode.sh "Urgent: Server-2 disk at 94%, need cleanup"
```

Session Management

Both agents run in named tmux sessions. The naming convention matters — it's how scripts find the right pane.

```
# Start Claude Code in a named session
tmux new-session -d -s claude-code -x 220 -y 50
tmux send-keys -t claude-code "claude" Enter

# Start OpenClaw in its own session
tmux new-session -d -s openclaw -x 220 -y 50
tmux send-keys -t openclaw "openclaw start" Enter

# List active sessions
tmux list-sessions

# Attach to a session to observe
tmux attach -t claude-code

# Send a task to Claude Code without attaching
```

```
tmux send-keys -t claude-code:0.0 "Check ~/.openclaw/workspace/inbox/from-openclaw.md" Enter
```

Spawning Sub-Agents: The Fast-Track Protocol

When to Spawn a Sub-Agent

Not every task warrants a sub-agent. Spawning adds overhead: initialization time, context setup, result collection. The rule we settled on:

Spawn a sub-agent if: - Task estimated time > 10 seconds - Task requires heavy tool use (many file reads, network calls, build operations) - Task is independent and parallelizable - Task might fill the current agent's context window

Handle inline if: - Simple lookup or calculation - Single tool call - Response needed immediately for the next step

```python
# fast-track-decision.py
# Logic for deciding when to spawn vs inline

from enum import Enum

class ExecutionMode(Enum):
    INLINE = "inline"
    SUB_AGENT = "sub_agent"

def decide_execution_mode(task: dict) -> ExecutionMode:
    """
    Decide whether to execute inline or spawn a sub-agent.
    """
    estimated_seconds = task.get("estimated_seconds", 0)
    tool_calls_expected = task.get("tool_calls_expected", 0)
    has_file_ops = task.get("has_file_operations", False)
    is_blocking = task.get("blocks_other_tasks", True)

    # Fast-track triggers
    if estimated_seconds > 10:
        return ExecutionMode.SUB_AGENT
    if tool_calls_expected > 5:
        return ExecutionMode.SUB_AGENT
    if has_file_ops and tool_calls_expected > 2:
        return ExecutionMode.SUB_AGENT

    # Inline: simple, fast, needs result immediately
    return ExecutionMode.INLINE
```

Sub-Agent Spawn Patterns

```
# agent-spawner.py
# Patterns for spawning sub-agents with proper lifecycle management

import subprocess
import json
import time
import os
from pathlib import Path

def spawn_claude_subagent(
    task_description: str,
    model: str = "claude-opus-4-6",
    working_dir: str = None,
    context_files: list[str] = None,
    timeout_minutes: int = 30,
) -> dict:
    """
    Spawn a Claude sub-agent for a scoped task.
    Always uses --cleanup delete to avoid session accumulation.
    """
    context_parts = []
    if context_files:
        for f in context_files:
            if Path(f).exists():
                context_parts.append(f"Context file: {f}")

    full_prompt = f"""
{task_description}

Working directory: {working_dir or os.getcwd()}
{chr(10).join(context_parts)}

When complete, write a brief summary of what was done to:
~/.openclaw/workspace/inbox/from-claudecode.md

Include: status (SUCCESS/FAILED), files modified, and any blockers.
"""

    cmd = [
        "claude",
        "--model", model,
        "--print",                          # Non-interactive mode
        "--cleanup", "delete",              # ALWAYS: clean up session after
        "--max-turns", "50",               # Prevent infinite loops
        full_prompt,
```

```
    ]

    if working_dir:
        cmd.extend(["--cwd", working_dir])

    start_time = time.time()

    try:
        result = subprocess.run(
            cmd,
            capture_output=True,
            text=True,
            timeout=timeout_minutes * 60,
        )

        elapsed = time.time() - start_time
        return {
            "success": result.returncode == 0,
            "output": result.stdout,
            "error": result.stderr,
            "elapsed_seconds": round(elapsed, 1),
            "model": model,
        }

    except subprocess.TimeoutExpired:
        return {
            "success": False,
            "error": f"Sub-agent timed out after {timeout_minutes} minutes",
            "elapsed_seconds": timeout_minutes * 60,
            "model": model,
        }


def spawn_gemini_subagent(task_description: str, model: str = "gemini-2.0-flash") -> dict:
    """
    Spawn a Gemini sub-agent for content/research tasks.
    Gemini: OK for content, research, analysis.
    NOT OK for infrastructure changes.
    """
    # Implementation varies by Gemini CLI setup
    cmd = ["gemini", "--model", model, "--print", task_description]

    result = subprocess.run(cmd, capture_output=True, text=True, timeout=300)
    return {
        "success": result.returncode == 0,
        "output": result.stdout,
        "model": model,
```

```
    }
```

The cleanup: delete Rule

Every sub-agent spawn must include `--cleanup delete`. Without it, sub-agent sessions accumulate. After a few days of heavy orchestration, you end up with dozens of orphaned sessions consuming memory and cluttering the session list.

We learned this after a weekend of heavy batch processing left 47 orphaned sessions.

```
# Clean up orphaned sessions manually (after forgetting --cleanup delete)
claude --list-sessions | grep "sub-agent" | awk '{print $1}' | xargs -I{} claude --delete-sessi

# Or nuclear option
claude --list-sessions | grep -v "main\|claude-code" | awk '{print $1}' | xargs -I{} claude --
```

Model Policy: Which Agent for Which Task

The Hard Rule

We made one policy non-negotiable after a painful incident: **infrastructure tasks require Claude Opus 4.6. No exceptions.**

The incident: we spawned a Gemini sub-agent to fix a failing systemd service. Gemini tried. It ran `systemctl --user restart service-name`. The service was a system service, not a user service. The command succeeded (no error, user-level services just don't exist). The agent reported success. The actual service was still down. We didn't discover this until a monitoring alert fired 20 minutes later.

```
# model-policy.py
# Enforce model selection policy for sub-agents

from enum import Enum

class TaskCategory(Enum):
    INFRASTRUCTURE = "infrastructure"    # systemd, networking, DB, server config
    CODE_REVIEW = "code_review"          # Analyze and critique code
    CODE_WRITING = "code_writing"        # Implement features, fix bugs
    CONTENT = "content"                  # Blog posts, docs, copy
    RESEARCH = "research"                # Web search, analysis, summarization
    DATA_ANALYSIS = "data_analysis"      # Query results, log analysis

# Model policy - do not modify without team discussion
MODEL_POLICY = {
    TaskCategory.INFRASTRUCTURE: {
        "model": "claude-opus-4-6",
        "rationale": "Infrastructure errors are hard to detect and costly to recover",
        "override_allowed": False,
```

```
    },
    TaskCategory.CODE_REVIEW: {
        "model": "claude-opus-4-6",
        "rationale": "Code review quality directly affects production reliability",
        "override_allowed": False,
    },
    TaskCategory.CODE_WRITING: {
        "model": "claude-sonnet-4-6",  # Balance quality/cost
        "rationale": "Complex enough to need strong model, not always Opus",
        "override_allowed": True,
    },
    TaskCategory.CONTENT: {
        "model": "gemini-2.0-flash",
        "rationale": "Content quality doesn't have hard failure modes",
        "override_allowed": True,
    },
    TaskCategory.RESEARCH: {
        "model": "gemini-2.0-flash",
        "rationale": "Research is low-risk, Gemini handles it well",
        "override_allowed": True,
    },
    TaskCategory.DATA_ANALYSIS: {
        "model": "gemini-2.0-flash",
        "rationale": "Structured analysis task, model matters less than context",
        "override_allowed": True,
    },
}

def get_model_for_task(category: TaskCategory, requested_model: str = None) -> str:
    policy = MODEL_POLICY[category]

    if requested_model and policy["override_allowed"]:
        return requested_model

    if requested_model and not policy["override_allowed"]:
        print(
            f"WARNING: Model override '{requested_model}' rejected for {category.value}. "
            f"Policy requires: {policy['model']}. Reason: {policy['rationale']}"
        )

    return policy["model"]
```

Why Gemini Sub-Agents Can't Fix System Issues

The systemctl mistake wasn't a Gemini intelligence problem. It was a context problem. Gemini, by default, doesn't know:

- Whether a service is a user service or a system service

- What the actual systemd unit file looks like
- Whether the service has been enabled vs started
- The difference between `journalctl -u` and `journalctl --user-unit`

Claude Opus 4.6, given the same task, asks clarifying questions or checks the service configuration before acting. It's not that Gemini can't produce the right answer — it's that infrastructure work requires a model that knows when it doesn't know enough to proceed.

Sub-Agent Failure Modes

Failure Mode 1: The 200K Context Limit

Sub-agents, like primary agents, have context limits. A sub-agent tasked with "migrate the entire database schema" will hit the limit before completing if the schema is large and the migration is complex.

The symptom: the sub-agent starts producing shorter and shorter responses, stops using tools, and eventually truncates its output.

The fix: scope tasks more narrowly. Instead of "migrate the database," spawn: - Sub-agent A: migrate tables 1-20 - Sub-agent B: migrate tables 21-40 - Sub-agent C: update foreign key constraints

```
# task-splitter.py
# Split large tasks into sub-agent-sized chunks

def split_schema_migration(tables: list[str], chunk_size: int = 20) -> list[dict]:
    """Split a large schema migration into manageable sub-tasks."""
    chunks = []
    for i in range(0, len(tables), chunk_size):
        chunk = tables[i:i + chunk_size]
        chunks.append({
            "task_id": f"migrate-tables-{i//chunk_size + 1}",
            "tables": chunk,
            "description": f"Migrate tables {i+1} to {min(i+chunk_size, len(tables))}: {', '.j
            "depends_on": f"migrate-tables-{i//chunk_size}" if i > 0 else None,
        })
    return chunks


def estimate_context_usage(task_description: str, context_files: list[str]) -> int:
    """
    Rough estimate of context tokens a task will use.
    Returns estimated token count.
    """
    # Rough heuristic: 1 token per 4 characters
    desc_tokens = len(task_description) // 4

    file_tokens = 0
```

```
    for f in context_files:
        try:
            file_size = Path(f).stat().st_size
            file_tokens += file_size // 4
        except FileNotFoundError:
            pass

    # Add buffer for tool call overhead
    overhead = 50_000  # Typical tool call overhead for a complex task

    return desc_tokens + file_tokens + overhead


def should_split_task(task: dict, limit: int = 150_000) -> bool:
    """Determine if a task needs splitting before spawning."""
    estimated = estimate_context_usage(
        task.get("description", ""),
        task.get("context_files", []),
    )
    if estimated > limit:
        print(f"Task context estimate {estimated:,} tokens exceeds limit {limit:,}. Splitting.
        return True
    return False
```

Failure Mode 2: Wrong Service Level (User vs System Systemd)

The systemctl failure from the model policy section is a category of failure worth documenting explicitly. AI agents don't always know the difference between:

```
systemctl status service-name          # System service (requires sudo)
systemctl --user status service-name   # User service (current user's context)
```

A system service running as `root` or a dedicated service user will not appear in `systemctl --user` output. An agent checking the wrong one will report "service not found" or "service inactive" when the service is actually running fine at the system level.

Prevention: always specify the service level in the task description.

```
BAD:  "Check if nginx is running and restart it if not"
GOOD: "Check if nginx is running as a SYSTEM service (sudo systemctl status nginx)
       and restart it if not. Do NOT use --user flag."
```

Failure Mode 3: Silent Failure on Missing Directory

This is the one that stung most: a Firecrawl deployment where the sub-agent was tasked with deploying configuration files to `/opt/firecrawl/config/`.

The directory didn't exist.

The sub-agent tried to write the files, got a "No such file or directory" error, and — instead of

failing loudly — logged it as a warning, marked the task complete in its output, and exited with code 0.

The orchestrator saw "SUCCESS" and moved on to the next task (starting the Firecrawl service). The service failed to start because the config files weren't there.

```
# Debugging the incident after the fact
# The sub-agent output was technically correct - it reported what happened
# But "what happened" was buried in a wall of text

grep -n "No such file\|WARN\|ERROR\|failed" /tmp/subagent-firecrawl-deploy.log
# Line 847: WARN: Could not write /opt/firecrawl/config/app.json: No such file or directory
# Line 848: Skipping config file deployment, service may not start correctly
# Line 1203: Task complete. 3/4 steps succeeded.
```

The fix is validation. Before a sub-agent declares success, it must verify its own work.

```
# sub-agent-validator.py
# Validation patterns sub-agents should use before reporting success

import os
from pathlib import Path

class TaskValidator:
    """Validation checks sub-agents should run before reporting completion."""

    @staticmethod
    def verify_files_written(expected_files: list[str]) -> tuple[bool, list[str]]:
        """Verify that files were actually written."""
        missing = []
        for filepath in expected_files:
            path = Path(filepath)
            if not path.exists():
                missing.append(filepath)
            elif path.stat().st_size == 0:
                missing.append(f"{filepath} (empty)")
        return len(missing) == 0, missing

    @staticmethod
    def verify_directory_exists(dirpath: str) -> bool:
        """Verify a directory exists before trying to write to it."""
        return Path(dirpath).is_dir()

    @staticmethod
    def verify_service_running(service_name: str, user_level: bool = False) -> bool:
        """Verify a systemd service is actually running."""
        import subprocess
        cmd = ["systemctl"]
        if user_level:
```

```
            cmd.append("--user")
        cmd.extend(["is-active", "--quiet", service_name])
        result = subprocess.run(cmd, capture_output=True)
        return result.returncode == 0

    @staticmethod
    def pre_flight_checks(task: dict) -> tuple[bool, list[str]]:
        """Run pre-flight checks before starting task."""
        errors = []

        # Check target directories exist before writing
        for target_dir in task.get("target_directories", []):
            if not Path(target_dir).exists():
                errors.append(f"Target directory does not exist: {target_dir}")
                # Try to create it
                try:
                    Path(target_dir).mkdir(parents=True, exist_ok=True)
                    errors[-1] += " (created)"
                    errors.pop()  # Remove error if creation succeeded
                except PermissionError:
                    errors[-1] += " (permission denied, cannot create)"

        # Check required files exist
        for required_file in task.get("required_files", []):
            if not Path(required_file).exists():
                errors.append(f"Required file missing: {required_file}")

        return len(errors) == 0, errors
```

The rule we now enforce: Sub-agent task descriptions must include an explicit verification step.

```
TASK: Deploy Firecrawl configuration files

Steps:
1. Ensure /opt/firecrawl/config/ directory exists (create if missing)
2. Write app.json, redis.json, and env.json to that directory
3. VERIFY: confirm all three files exist and are non-empty
4. Only report SUCCESS if all three files are verified present

If directory creation fails due to permissions, report FAILED with details.
Do NOT report success if any file is missing.
```

Provider Fallback Chain

The Three-Provider Setup

We run three LLM provider configurations in priority order:

`CLIProxyAPI → Anthropic Native → ZAI`

- **CLIProxyAPI:** Our internal proxy that routes to Anthropic, adds request logging, enforces rate limits, and handles billing aggregation
- **Anthropic Native:** Direct Anthropic API — fallback if the proxy is down
- **ZAI:** Third-party API service — fallback if Anthropic native is having issues

```
# provider-fallback-chain.py
# LLM provider with automatic fallback on failure

import time
import os
from dataclasses import dataclass
from typing import Optional

@dataclass
class ProviderConfig:
    name: str
    base_url: str
    api_key_env: str
    model_map: dict  # Map canonical model names to provider-specific names
    priority: int    # Lower = higher priority

PROVIDERS = [
    ProviderConfig(
        name="cli-proxy",
        base_url=os.environ.get("CLI_PROXY_URL", "http://10.8.0.5:4000"),
        api_key_env="CLI_PROXY_API_KEY",
        model_map={
            "claude-opus-4-6": "claude-opus-4-6",
            "claude-sonnet-4-6": "claude-sonnet-4-6",
            "claude-haiku-4-5": "claude-haiku-4-5",
        },
        priority=1,
    ),
    ProviderConfig(
        name="anthropic-native",
        base_url="https://api.anthropic.com",
        api_key_env="ANTHROPIC_API_KEY",
        model_map={
            "claude-opus-4-6": "claude-opus-4-6",
            "claude-sonnet-4-6": "claude-sonnet-4-6",
            "claude-haiku-4-5": "claude-haiku-4-5",
        },
        priority=2,
    ),
    ProviderConfig(
        name="zai",
        base_url="https://api.zai.ai/v1",
```

```
        api_key_env="ZAI_API_KEY",
        model_map={
            "claude-opus-4-6": "claude-3-opus",  # ZAI uses different naming
            "claude-sonnet-4-6": "claude-3-5-sonnet",
            "claude-haiku-4-5": "claude-3-haiku",
        },
        priority=3,
    ),
]


class FallbackLLMClient:
    def __init__(self):
        self.providers = sorted(PROVIDERS, key=lambda p: p.priority)
        self.failed_until: dict[str, float] = {}  # provider_name → timestamp

    def _is_available(self, provider: ProviderConfig) -> bool:
        failed_time = self.failed_until.get(provider.name, 0)
        return time.time() > failed_time

    def _mark_failed(self, provider: ProviderConfig, cooldown: int = 300):
        self.failed_until[provider.name] = time.time() + cooldown
        print(f"Provider {provider.name} marked failed for {cooldown}s")

    def complete(self, model: str, messages: list, **kwargs) -> Optional[str]:
        for provider in self.providers:
            if not self._is_available(provider):
                continue

            api_key = os.environ.get(provider.api_key_env)
            if not api_key:
                continue

            provider_model = provider.model_map.get(model, model)

            try:
                response = self._call_provider(
                    provider=provider,
                    model=provider_model,
                    messages=messages,
                    api_key=api_key,
                    **kwargs,
                )
                return response

            except Exception as e:
                error_str = str(e)
```

```
                if any(code in error_str for code in ["503", "502", "529", "overloaded"]):
                    # Provider overloaded - short cooldown
                    self._mark_failed(provider, cooldown=120)
                elif "401" in error_str or "403" in error_str:
                    # Auth failure - longer cooldown, check credentials
                    self._mark_failed(provider, cooldown=3600)
                    print(f"Auth failure for {provider.name}: check {provider.api_key_env}")
                else:
                    # Unknown error - brief cooldown, try next
                    self._mark_failed(provider, cooldown=60)
                    print(f"Error with {provider.name}: {e}")

        return None  # All providers failed

    def _call_provider(self, provider, model, messages, api_key, **kwargs) -> str:
        # Use anthropic SDK or httpx depending on provider
        import anthropic

        client = anthropic.Anthropic(
            api_key=api_key,
            base_url=provider.base_url if provider.name != "anthropic-native" else None,
        )
        response = client.messages.create(
            model=model,
            messages=messages,
            max_tokens=kwargs.get("max_tokens", 4096),
        )
        return response.content[0].text
```

Cross-Server Agent Communication

The VPN-Based Mesh

Our agents don't all run on the same server. OpenClaw runs on the primary workstation. Some batch jobs run on dedicated compute servers. The monitoring agent runs on Server-3.

Communication happens over the WireGuard VPN mesh. All servers are on `10.8.0.0/24`. Agents find each other by VPN IP.

```
# /etc/wireguard/wg0.conf (simplified)
[Interface]
PrivateKey = <redacted>
Address = 10.8.0.1/24
ListenPort = 51820

# Workstation (OpenClaw)
```

```
[Peer]
PublicKey = <workstation-pubkey>
AllowedIPs = 10.8.0.2/32

# Server-3 (Cognee, monitoring)
[Peer]
PublicKey = <server3-pubkey>
AllowedIPs = 10.8.0.12/32

# Compute server (batch jobs)
[Peer]
PublicKey = <compute-pubkey>
AllowedIPs = 10.8.0.20/32
```

For cross-server task delegation, agents use a lightweight HTTP API rather than shared filesystem (the filesystem isn't shared across physical servers):

```
# cross-server-agent-api.py
# Simple HTTP API for cross-server task delegation

from fastapi import FastAPI, HTTPException, Header
from pydantic import BaseModel
import subprocess
import asyncio
import os
import hmac
import hashlib

app = FastAPI(title="Agent Task API")
SHARED_SECRET = os.environ["AGENT_API_SECRET"]

class TaskRequest(BaseModel):
    task: str
    model: str = "claude-opus-4-6"
    timeout_minutes: int = 30
    working_dir: str = None
    priority: str = "normal"

class TaskResponse(BaseModel):
    task_id: str
    status: str
    output: str = None
    error: str = None

def verify_signature(payload: str, signature: str) -> bool:
    """Verify HMAC signature on incoming requests."""
    expected = hmac.new(
        SHARED_SECRET.encode(),
```

```
        payload.encode(),
        hashlib.sha256
    ).hexdigest()
    return hmac.compare_digest(expected, signature)

@app.post("/tasks", response_model=TaskResponse)
async def create_task(
    request: TaskRequest,
    x_signature: str = Header(...),
):
    # Verify the request came from a trusted agent
    payload = f"{request.task}:{request.model}"
    if not verify_signature(payload, x_signature):
        raise HTTPException(status_code=401, detail="Invalid signature")

    task_id = f"task-{int(asyncio.get_event_loop().time())}"

    # Spawn sub-agent asynchronously
    asyncio.create_task(run_subagent(task_id, request))

    return TaskResponse(task_id=task_id, status="accepted")


async def run_subagent(task_id: str, request: TaskRequest):
    """Run a sub-agent task and store results."""
    cmd = [
        "claude", "--model", request.model,
        "--print", "--cleanup", "delete",
        request.task,
    ]

    if request.working_dir:
        cmd.extend(["--cwd", request.working_dir])

    result = await asyncio.create_subprocess_exec(
        *cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE,
    )

    stdout, stderr = await asyncio.wait_for(
        result.communicate(),
        timeout=request.timeout_minutes * 60
    )

    # Store result for retrieval
    results_store[task_id] = {
```

```
        "status": "complete" if result.returncode == 0 else "failed",
        "output": stdout.decode(),
        "error": stderr.decode(),
    }

results_store = {}

@app.get("/tasks/{task_id}", response_model=TaskResponse)
async def get_task_result(task_id: str):
    result = results_store.get(task_id)
    if not result:
        return TaskResponse(task_id=task_id, status="pending")
    return TaskResponse(task_id=task_id, **result)
```

Agent Mesh Healing: Flock-Based Distributed Coordination

The Problem: Two Agents, Same Task

In a multi-agent setup, two agents can independently decide to work on the same task. This is fine for read-only tasks. It's catastrophic for write operations — two agents simultaneously writing configuration files, restarting services, or running migrations.

We use filesystem locking via `flock` to coordinate exclusive access to resources.

```
#!/bin/bash
# agent-lock.sh
# Acquire an exclusive lock before performing write operations
# Usage: agent-lock.sh <lock-name> <command>

LOCK_NAME="${1:-default}"
LOCK_FILE="/tmp/agent-locks/${LOCK_NAME}.lock"
LOCK_TIMEOUT="${AGENT_LOCK_TIMEOUT:-300}"  # 5 minutes default

mkdir -p /tmp/agent-locks

# Try to acquire lock with timeout
if flock --exclusive --timeout "$LOCK_TIMEOUT" 9; then
    echo "Lock acquired: $LOCK_NAME"
    # Run the command
    shift  # Remove lock-name from args
    "$@"
    EXIT_CODE=$?
    echo "Command complete, releasing lock: $LOCK_NAME"
    exit $EXIT_CODE
else
    echo "ERROR: Could not acquire lock '$LOCK_NAME' within ${LOCK_TIMEOUT}s"
```

```
    echo "Another agent may be working on this resource"
    exit 1
fi 9>"$LOCK_FILE"
```

```
# distributed-lock.py
# Python context manager for distributed agent locking

import fcntl
import os
import time
from contextlib import contextmanager
from pathlib import Path

LOCK_DIR = Path("/tmp/agent-locks")
LOCK_DIR.mkdir(exist_ok=True)

@contextmanager
def agent_lock(resource_name: str, timeout: int = 300):
    """
    Context manager for exclusive agent access to a resource.

    Usage:
        with agent_lock("database-migration"):
            run_migration()
    """
    lock_file = LOCK_DIR / f"{resource_name}.lock"
    lock_fd = open(lock_file, "w")

    start = time.time()
    acquired = False

    try:
        while time.time() - start < timeout:
            try:
                fcntl.flock(lock_fd, fcntl.LOCK_EX | fcntl.LOCK_NB)
                acquired = True
                lock_fd.write(f"Locked by PID {os.getpid()} at {time.time()}\n")
                lock_fd.flush()
                break
            except IOError:
                elapsed = time.time() - start
                print(f"Waiting for lock '{resource_name}' ({elapsed:.0f}s/{timeout}s)...")
                time.sleep(5)

        if not acquired:
            raise TimeoutError(
                f"Could not acquire lock '{resource_name}' within {timeout}s. "
```

```
                f"Another agent may be holding it."
            )

        print(f"Lock acquired: {resource_name}")
        yield

    finally:
        if acquired:
            fcntl.flock(lock_fd, fcntl.LOCK_UN)
            print(f"Lock released: {resource_name}")
        lock_fd.close()


# Usage in a sub-agent task
async def deploy_configuration(config_files: dict):
    with agent_lock("firecrawl-config-deployment", timeout=120):
        # Only one agent can deploy Firecrawl config at a time
        for filename, content in config_files.items():
            target = Path("/opt/firecrawl/config") / filename
            target.parent.mkdir(parents=True, exist_ok=True)
            target.write_text(content)
            print(f"Written: {target}")
```

Self-Healing: Detecting and Recovering from Stalled Agents

```
# agent-health-monitor.py
# Monitor sub-agent health and recover stalled tasks

import time
import json
import subprocess
from pathlib import Path
from datetime import datetime, timedelta

TASK_LOG = Path("/tmp/agent-tasks.jsonl")
STALL_THRESHOLD_MINUTES = 30  # Task running longer than this is suspect

def log_task_start(task_id: str, task_description: str, pid: int):
    """Log task start for monitoring."""
    entry = {
        "task_id": task_id,
        "event": "start",
        "description": task_description[:100],
        "pid": pid,
        "timestamp": datetime.utcnow().isoformat(),
    }
    with open(TASK_LOG, "a") as f:
```

```
        f.write(json.dumps(entry) + "\n")

def log_task_end(task_id: str, status: str):
    """Log task completion."""
    entry = {
        "task_id": task_id,
        "event": "end",
        "status": status,
        "timestamp": datetime.utcnow().isoformat(),
    }
    with open(TASK_LOG, "a") as f:
        f.write(json.dumps(entry) + "\n")

def find_stalled_tasks() -> list[dict]:
    """Find tasks that have been running longer than threshold."""
    if not TASK_LOG.exists():
        return []

    # Build task state from log
    tasks = {}
    with open(TASK_LOG) as f:
        for line in f:
            try:
                entry = json.loads(line)
                task_id = entry["task_id"]
                if entry["event"] == "start":
                    tasks[task_id] = entry
                elif entry["event"] == "end":
                    tasks.pop(task_id, None)
            except json.JSONDecodeError:
                continue

    # Check which running tasks are stalled
    threshold = datetime.utcnow() - timedelta(minutes=STALL_THRESHOLD_MINUTES)
    stalled = []

    for task_id, task in tasks.items():
        start_time = datetime.fromisoformat(task["timestamp"])
        if start_time < threshold:
            # Check if process is still running
            pid = task.get("pid")
            if pid:
                try:
                    import psutil
                    proc = psutil.Process(pid)
                    if proc.is_running():
                        stalled.append({**task, "running_minutes": (datetime.utcnow() - start_t
```

```
                except (psutil.NoSuchProcess, psutil.AccessDenied):
                    # Process is gone but task wasn't logged as ended
                    log_task_end(task_id, "abandoned")

    return stalled

def recover_stalled_task(task: dict):
    """Attempt to recover a stalled task."""
    pid = task.get("pid")
    task_id = task["task_id"]

    print(f"Recovering stalled task {task_id} (PID {pid}, running {task['running_minutes']}min)

    # Kill the stalled process
    if pid:
        try:
            subprocess.run(["kill", "-TERM", str(pid)], check=False)
            time.sleep(5)
            subprocess.run(["kill", "-KILL", str(pid)], check=False)
        except Exception as e:
            print(f"Could not kill PID {pid}: {e}")

    # Release any locks this task was holding
    for lock_file in Path("/tmp/agent-locks").glob("*.lock"):
        try:
            import fcntl
            with open(lock_file) as f:
                content = f.read()
                if str(pid) in content:
                    lock_file.unlink()
                    print(f"Released lock: {lock_file.name}")
        except Exception:
            pass

    log_task_end(task_id, "recovered")
    print(f"Task {task_id} recovered. Recommend re-queuing: {task['description']}")
```

Sub-Agent Best Practices: The Runbook

After a year of running multi-agent infrastructure, here's the operational runbook we hand to anyone new to the system.

Do's

```
[ ] Always include --cleanup delete when spawning sub-agents
```

```
[ ] Specify model explicitly - never rely on defaults for infra tasks
[ ] Include a verification step in every write task
[ ] Use flock or equivalent before any write operation
[ ] Set explicit timeouts - never let sub-agents run indefinitely
[ ] Log task start and end with task IDs for health monitoring
[ ] Include explicit working directory in task descriptions
[ ] For system services, specify system vs user level in the task
[ ] Test sub-agent tasks on staging before running in production
[ ] Scope tasks to fit in ~100K tokens of context (leave 50K headroom)
```

Don'ts

```
[ ] Don't assume sub-agent "success" without verifying outputs
[ ] Don't spawn sub-agents for tasks that can complete in <10s
[ ] Don't use Gemini for infrastructure changes
[ ] Don't let sub-agents accumulate - check session list regularly
[ ] Don't share datasets across agents without explicit isolation
[ ] Don't hardcode server addresses - use VPN hostnames or env vars
[ ] Don't run migrations or schema changes without a flock lock
[ ] Don't trust sub-agent output that doesn't include explicit verification
[ ] Don't spawn sub-agents in loops without rate limiting
```

Task Description Template

```
## Task: [Brief title]

**Model:** claude-opus-4-6 [or specify]
**Working Directory:** /path/to/workdir
**Service Level:** system [or user, for systemd tasks]
**Timeout:** 20 minutes

### Pre-flight Checks
Before starting, verify:
- [ ] [Directory X] exists (create if missing with proper permissions)
- [ ] [Service Y] is currently [running/stopped]
- [ ] [File Z] is readable

### Steps
1. [Specific, unambiguous step]
2. [Specific, unambiguous step]
3. [Specific, unambiguous step]

### Verification
After completing all steps, confirm:
- [ ] [Expected file] exists and is non-empty
- [ ] [Expected service] is active (use: sudo systemctl is-active [service])
- [ ] [Expected endpoint] responds (use: curl -f http://...)
```

```
### Report
Write to ~/.openclaw/workspace/inbox/from-claudecode.md:
- Status: SUCCESS or FAILED
- What was done
- Any issues encountered
- Files modified (with paths)
```

Lessons Learned

"Success" from a sub-agent means the agent finished, not that the task succeeded. Build explicit verification into every task. Treat the agent's report as a hint, not a fact.

Context limit failures are silent. A sub-agent that hits the context limit doesn't crash — it degrades. It produces shorter responses, skips steps, and may report success on an incomplete task. Scope tasks to use less than two-thirds of the context window.

Model selection is a policy decision, not a preference. Enforce it programmatically. The human who understands why Gemini can't be trusted for system-level service management may not be on call when the alert fires.

Cleanup is not optional. –cleanup delete, flock releases, lock file cleanup — all of it. Orphaned sessions and stale locks compound over time into operational incidents.

The shared workspace is low-tech and reliable. We evaluated several message brokers for agent communication. We shipped the shared directory approach. Flat files survive server reboots, don't have broker availability issues, and are trivially inspectable by any tool.

Cross-server communication needs authentication. Agents running on different servers communicating over VPN still need request signing. VPN provides network-level isolation; HMAC signing provides application-level identity. Both are necessary.

Multi-agent orchestration is not a solved problem. Every system has its own failure modes, its own optimal task boundaries, its own model policy. What works for us — tmux + shared workspace + flock + explicit verification — works because it matches our specific infrastructure. Your setup will be different.

The universal principles: fail loudly, verify explicitly, scope narrowly, and never trust a sub-agent's report without checking its work.

End of Part 4: Advanced Topics.

Appendix A covers reference architectures. Appendix B covers the complete tooling stack used throughout this book.

PART 5: THE FUTURE

Chapter 15: Scaling Agent Systems

> "The first agent is a proof of concept. The tenth is an operational nightmare. The hundredth is either a competitive advantage or a burning pile of money — usually both."

Introduction

When I deployed my first AI agent in production, I thought I was done with the hard part. I had a working agent, a stable server, and a nervous optimism that things would just… work. Six months later, I had seventeen agents running across four servers, a Patroni cluster that had failed over twenty-two times in a weekend, and a cloud bill that made my hands shake.

Scaling agent systems is not like scaling traditional microservices. The failure modes are weirder. The costs are less predictable. And the debugging is more philosophical — "why did the agent decide that rebooting the database was the correct response to a slow query?" is not a question you expect to ask at 2 AM.

This chapter is the one I wish I'd had before I started scaling. It covers the technical realities: database high-availability, load balancing, memory systems, and cost management. But it also covers the operational realities that the whitepapers never mention — like why I eventually deleted every automated cron job in the system, and why that was the right call.

Part 1: From One Agent to an Agent Mesh

The Single-Agent Phase (Days 1–30)

Your first agent is simple. It runs on one server, uses one model, and has one job. You can reason about it in your head. When it breaks, you know where to look.

This phase lasts until the agent works well enough that you want it to do more. That is the trap.

The correct instinct — "this works, let's add more capability" — leads directly to the incorrect architecture: a single agent doing ten jobs badly instead of one job well.

```
# Single-agent architecture (simple but brittle)

          Server-1
```

```
  Agent-Alpha
- Infra monitoring
- Log analysis
- Alert routing
- Incident reports
- Cost tracking
```

The agent becomes slow because it is doing too much. Context windows fill up. The model starts making worse decisions because the prompt is now fifteen thousand tokens of mixed context. You add more memory. It gets slower.

The inflection point: When an agent's average response latency exceeds 15 seconds and you catch yourself thinking "I'll just add more instructions to fix this," it is time to split.

Designing the Agent Mesh (Days 30–90)

A mesh is not just multiple agents — it is a deliberate topology where agents have clear ownership boundaries and well-defined communication protocols.

The minimal viable mesh has three roles:

1. **Orchestrator** — receives requests, decomposes tasks, delegates to specialists, aggregates results
2. **Specialists** — deep capability in one domain (infra, code, data, etc.)
3. **Observers** — passive monitoring, no write access, report-only

```
# Agent mesh topology (3-agent minimum)

                    Server-1 (Orchestrator)

                OpenClaw (Orchestrator)
    - Request routing
    - Task decomposition
    - Result aggregation


               HTTP/tmux IPC       HTTP/tmux IPC

    Server-2                     Server-3

    Claude Code                Monitoring Agent
    (Specialist)               (Observer)
    - Code gen                 - Prometheus
    - Reviews                  - Alertmanager
    - Testing                  - No write access
```

Communication between agents: I tried three approaches before settling on one that works.

- **REST APIs** — clean but adds latency and requires each agent to run an HTTP server. Overhead was ~200ms per hop, which compounds badly in chains.
- **Message queues (Redis Streams)** — better for async tasks, but adds infrastructure complexity and another thing to monitor.
- **tmux-based IPC** — not what you expect to see recommended in a book, but it works. Each agent runs in a named tmux session. The orchestrator sends commands by writing to a shared file that the target agent polls, or by using `tmux send-keys`. Latency is under 50ms for local calls. No extra infrastructure.

```
# Sending a task from Orchestrator to Claude Code agent via tmux IPC
send_agent_task() {
  local target_session="$1"
  local task_payload="$2"
  local inbox_file="$HOME/.agents/inbox/${target_session}.json"

  echo "$task_payload" > "$inbox_file"
  tmux send-keys -t "$target_session" \
    "cat $inbox_file | process_task" Enter
}

# Claude Code agent polling loop
poll_inbox() {
  local inbox_file="$HOME/.agents/inbox/claude-code.json"
  while true; do
    if [[ -f "$inbox_file" ]]; then
      task=$(cat "$inbox_file")
      rm "$inbox_file"
      process_task "$task"
    fi
    sleep 2
  done
}
```

This is not glamorous. It is also not fragile. tmux sessions survive SSH disconnects. Files do not require a running service. When everything else is on fire, you can still inspect the inbox directory with `ls`.

Mesh Governance: Who Owns What

The most common failure in multi-agent systems is not technical — it is ownership confusion. Two agents both try to fix the same alert. Neither agent checks whether the other already acted. The system double-provisions, double-sends, double-bills.

Implement a simple lease system before you have more than two agents:

```
# Simple distributed lease using PostgreSQL advisory locks
import psycopg2
import hashlib
```

```python
def acquire_task_lease(task_id: str, agent_id: str, ttl_seconds: int = 300) -> bool:
    """
    Returns True if this agent acquired the lease for task_id.
    Uses PostgreSQL advisory locks for atomicity.
    """
    lock_key = int(hashlib.md5(task_id.encode()).hexdigest()[:8], 16)

    with psycopg2.connect(DATABASE_URL) as conn:
        with conn.cursor() as cur:
            cur.execute("SELECT pg_try_advisory_lock(%s)", (lock_key,))
            acquired = cur.fetchone()[0]

            if acquired:
                # Record who holds the lease
                cur.execute("""
                    INSERT INTO agent_leases (task_id, agent_id, expires_at)
                    VALUES (%s, %s, NOW() + INTERVAL '%s seconds')
                    ON CONFLICT (task_id) DO UPDATE
                    SET agent_id = EXCLUDED.agent_id,
                        expires_at = EXCLUDED.expires_at
                """, (task_id, agent_id, ttl_seconds))
                conn.commit()

            return acquired
```

Part 2: Horizontal Scaling

When to Add a Server

Adding a server costs money. Not adding a server when you need one costs more money (in agent latency, missed SLAs, and your evenings). The decision criteria I use:

Signal	Threshold	Action
Agent queue depth	$>$ 10 pending tasks	Add specialist server
Average response latency	$>$ 20s sustained	Split workload
Memory pressure	$>$ 85% RAM consistently	Add server or reduce agents
Cost per task	$>$ \$0.05 average	Audit and optimize first

Do not add a server to solve a cost problem. You will have two servers and the same cost problem.

The Three-Server Baseline

By the time you have a production-grade mesh, you will likely converge on at least three servers:

```
Server-1 (Orchestrator / Control Plane)
  - Orchestrator agent
  - Database proxy (PgBouncer)
  - Reverse proxy (Traefik)
  - Monitoring stack (Prometheus, Grafana, Alertmanager)
  - RAM: 16GB minimum
  - CPU: 4 cores minimum

Server-2 (AI Workload / Hot Path)
  - High-frequency specialist agents
  - Fast storage (NVMe)
  - RAM: 32GB+ (agents are memory hungry)
  - CPU: 8+ cores
  - Note: GPU optional but rarely cost-effective for API-based agents

Server-3 (Database / Persistence Layer)
  - PostgreSQL primary (Patroni-managed)
  - Cognee knowledge graph
  - Backup agent
  - RAM: 16GB minimum (more if Cognee is heavily used)
  - Storage: High-IOPS, separate volume for WAL
```

My Server-3 is an Oracle ARM instance that has been running for approximately eight years without an unplanned outage. This is genuinely impressive. It is also slow for AI workloads — around 26 seconds per inference request when I tried running a local model on it. The ARM architecture runs database operations efficiently but struggles with the floating-point-heavy math that LLM inference requires. Lesson: ARM is excellent for databases, not for AI compute. Do not conflate uptime record with general-purpose fitness.

Load Balancing Agent Requests

For HTTP-based agent endpoints, Traefik is the right tool. It handles service discovery, TLS termination, and weighted routing without requiring you to write routing configuration by hand.

```yaml
# traefik/dynamic/agent-routing.yml
http:
  routers:
    agent-mesh:
      rule: "PathPrefix(`/agents`)"
      service: agent-pool
      middlewares:
        - agent-ratelimit
        - agent-retry

  services:
    agent-pool:
      loadBalancer:
        healthCheck:
          path: /health
```

```
          interval: "10s"
          timeout: "3s"
        servers:
          - url: "http://server-1:8080"
            weight: 1
          - url: "http://server-2:8080"
            weight: 3  # More capable server, higher weight

  middlewares:
    agent-ratelimit:
      rateLimit:
        average: 100
        burst: 50
        period: "1m"
        sourceCriterion:
          ipStrategy:
            depth: 1

    agent-retry:
      retry:
        attempts: 3
        initialInterval: "500ms"
```

For task-queue-based agents (the tmux IPC approach), load balancing is simpler — the orchestrator maintains a pool of available workers and assigns tasks round-robin with capability filtering:

```
class AgentPool:
    def __init__(self):
        self.agents: dict[str, AgentMeta] = {}
        self._lock = threading.Lock()

    def register(self, agent_id: str, capabilities: list[str], server: str):
        with self._lock:
            self.agents[agent_id] = AgentMeta(
                id=agent_id,
                capabilities=set(capabilities),
                server=server,
                active_tasks=0,
                last_health_check=time.time()
            )

    def assign(self, task: Task) -> str | None:
        """Returns agent_id of assigned agent, or None if no agent available."""
        with self._lock:
            candidates = [
                a for a in self.agents.values()
                if task.required_capability in a.capabilities
                and a.active_tasks < MAX_CONCURRENT_TASKS_PER_AGENT
```

```
            and time.time() - a.last_health_check < HEALTH_TIMEOUT
        ]

        if not candidates:
            return None

        # Least-loaded assignment
        chosen = min(candidates, key=lambda a: a.active_tasks)
        chosen.active_tasks += 1
        return chosen.id
```

Part 3: Database High-Availability — The Hard Lessons

This section is the one I most wish I had read before building. Database HA for agent systems is not dramatically different from HA for any other application — but agents interact with databases in ways that expose failure modes you would not encounter with human users.

Patroni + etcd: The Architecture

Patroni manages PostgreSQL failover. etcd provides the distributed consensus that Patroni uses to elect a primary. You need at least three etcd nodes for a proper quorum — with two nodes, any network partition causes both nodes to refuse to be primary, and your cluster goes read-only.

```
# Correct: 3 etcd nodes = tolerates 1 node failure
etcd-1 (Server-1)
etcd-2 (Server-2)       Patroni cluster
etcd-3 (Server-3)

# Incorrect: 2 etcd nodes = any partition = split-brain
etcd-1    etcd-2    (don't do this)
```

The etcd configuration for each node:

```
# /etc/etcd/etcd.conf.yml - node 1 example
name: etcd-server1
data-dir: /var/lib/etcd/data
wal-dir: /var/lib/etcd/wal

listen-peer-urls: http://10.0.1.1:2380
listen-client-urls: http://10.0.1.1:2379,http://127.0.0.1:2379

initial-advertise-peer-urls: http://10.0.1.1:2380
advertise-client-urls: http://10.0.1.1:2379

initial-cluster: >
  etcd-server1=http://10.0.1.1:2380,
  etcd-server2=http://10.0.1.2:2380,
```

```
  etcd-server3=http://10.0.1.3:2380

initial-cluster-token: ai-agents-production-cluster
initial-cluster-state: new

# Heartbeat and election tuning
heartbeat-interval: 100
election-timeout: 1000

# Compaction - critical for long-running clusters
auto-compaction-mode: periodic
auto-compaction-retention: "1h"
```

The 22-Failover Weekend: A Postmortem

On a Saturday morning, my monitoring started firing. Patroni was failing over. Then again. Then again. By Sunday evening, I had 22 failover events recorded in Patroni's history. The database was technically available — it kept electing a new primary — but every failover caused a 10-30 second outage for the agents. Background jobs were failing. Agents were retrying. Retries were generating tokens. Tokens were costing money.

Root cause: etcd configuration mismatch.

When I had added Server-3 to the cluster three weeks earlier, I had updated the `initial-cluster` string on servers 1 and 3 but had not updated it on server 2. The etcd documentation notes that `initial-cluster` is only used during bootstrapping, which is technically true — but it affects how nodes reconcile their peer lists after restarts. Server-2 had restarted during routine maintenance, rejoined with a stale peer list, and created a persistent inconsistency in how the three nodes saw each other.

Under normal load, this was invisible. Under the elevated load from a Saturday deployment, it surfaced as spurious leader elections.

The fix — step by step:

```
# Step 1: Identify the mismatched node
for server in server1 server2 server3; do
  echo "=== $server ==="
  ssh $server "etcdctl --endpoints=http://127.0.0.1:2379 member list"
done

# Step 2: Stop Patroni on all nodes FIRST (important: stop Patroni before etcd)
for server in server1 server2 server3; do
  ssh $server "sudo systemctl stop patroni"
done

# Step 3: Stop etcd on the mismatched node
ssh server2 "sudo systemctl stop etcd"

# Step 4: Clean etcd data on mismatched node (non-destructive to Postgres data)
```

```
ssh server2 "sudo rm -rf /var/lib/etcd/data /var/lib/etcd/wal"

# Step 5: Sync etcd config - every node must have identical initial-cluster
for server in server1 server2 server3; do
  ssh $server "sudo diff /etc/etcd/etcd.conf.yml /etc/etcd/etcd.conf.yml.reference"
done

# Step 6: Restart etcd in order - existing members first, new member last
ssh server1 "sudo systemctl start etcd"
ssh server3 "sudo systemctl start etcd"
sleep 10  # Let quorum establish
ssh server2 "sudo systemctl start etcd"

# Step 7: Verify cluster health before starting Patroni
etcdctl --endpoints=http://server1:2379,http://server2:2379,http://server3:2379 \
  endpoint health

# Step 8: Restart Patroni
for server in server1 server2 server3; do
  ssh $server "sudo systemctl start patroni"
  sleep 5  # Stagger to avoid thundering herd
done

# Step 9: Verify Patroni elected a single primary
patronictl -c /etc/patroni/patroni.yml list
```

Prevention: Add a configuration sync check to your deploy pipeline. Every etcd node should have an identical peer list. If they differ, the deploy fails.

```
#!/bin/bash
# etcd-config-check.sh - run before every deploy
NODES=(server1 server2 server3)
REFERENCE_HASH=$(ssh ${NODES[0]} "md5sum /etc/etcd/etcd.conf.yml | cut -d' ' -f1")

for node in "${NODES[@]:1}"; do
  NODE_HASH=$(ssh $node "md5sum /etc/etcd/etcd.conf.yml | cut -d' ' -f1")
  if [[ "$NODE_HASH" != "$REFERENCE_HASH" ]]; then
    echo "FAIL: etcd config mismatch on $node"
    echo "  Expected: $REFERENCE_HASH"
    echo "  Got:      $NODE_HASH"
    exit 1
  fi
done

echo "PASS: etcd config consistent across all nodes"
```

Patroni Configuration (Production-Hardened)

```
# /etc/patroni/patroni.yml
scope: ai-agents-prod
namespace: /patroni/
name: postgres-server1

restapi:
  listen: 0.0.0.0:8008
  connect_address: 10.0.1.1:8008

etcd3:
  hosts:
    - 10.0.1.1:2379
    - 10.0.1.2:2379
    - 10.0.1.3:2379

bootstrap:
  dcs:
    ttl: 30
    loop_wait: 10
    retry_timeout: 30
    maximum_lag_on_failover: 1048576  # 1MB - don't fail over if replica is too far behind
    postgresql:
      use_pg_rewind: true
      use_slots: true
      parameters:
        max_connections: 200
        shared_buffers: 4GB
        effective_cache_size: 12GB
        maintenance_work_mem: 512MB
        wal_level: replica
        hot_standby: "on"
        wal_keep_size: 1GB
        max_wal_senders: 10
        max_replication_slots: 10

postgresql:
  listen: 0.0.0.0:5432
  connect_address: 10.0.1.1:5432
  data_dir: /var/lib/postgresql/15/main
  bin_dir: /usr/lib/postgresql/15/bin
  pgpass: /tmp/pgpass0

  authentication:
    replication:
      username: replicator
```

```
      password: ${REPLICATION_PASSWORD}
    superuser:
      username: postgres
      password: ${POSTGRES_PASSWORD}

  parameters:
    unix_socket_directories: '/var/run/postgresql'
```

Backup Strategy: Three Repositories

For agent system databases, a single backup destination is insufficient. Agents read and write constantly. A corrupted backup that you discover during recovery is a worst-case scenario.

My production backup strategy uses three independent destinations:

```
Backup Target 1: Vietnix S3 (primary, local datacenter)
  - Full backup: daily at 02:00
  - WAL archiving: continuous
  - Retention: 30 days
  - Estimated cost: ~$8/month for 30GB

Backup Target 2: Oracle Object Storage (secondary, different cloud)
  - Full backup: daily at 03:00
  - WAL archiving: continuous
  - Retention: 14 days
  - Estimated cost: $0 (Oracle free tier covers this)

Backup Target 3: MinIO (self-hosted, on-premises)
  - Full backup: weekly
  - Retention: 90 days
  - Estimated cost: electricity + disk (~$2/month)
```

The numbers on compression are worth understanding. A full backup of the production database (agent conversations, knowledge graph metadata, task history, cost records) compressed as follows:

```
Raw PostgreSQL data directory:    561.7 MB
pg_basebackup (tar):              418.3 MB
pgBackRest (lz4 compressed):       30.2 MB
Compression ratio:                18.6:1
```

LZ4 compression on PostgreSQL data is remarkably effective because agent system databases contain a lot of repetitive JSON structure and repeated model names, tool names, and error messages. Always enable compression on your backup tool.

```
# /etc/pgbackrest/pgbackrest.conf
[global]
repo1-path=/var/lib/pgbackrest
repo1-retention-full=30
repo1-retention-diff=7
compress-type=lz4
```

```
compress-level=6
archive-async=y
archive-push-queue-max=4GiB

repo2-type=s3
repo2-path=/pgbackrest-prod
repo2-s3-bucket=vietnix-db-backups
repo2-s3-endpoint=s3.vietnix.vn
repo2-s3-region=ap-southeast-1
repo2-s3-key=${VIETNIX_S3_KEY}
repo2-s3-key-secret=${VIETNIX_S3_SECRET}

repo3-type=s3
repo3-path=/pgbackrest-prod
repo3-s3-bucket=oracle-db-backups
repo3-s3-endpoint=objectstorage.ap-singapore-1.oraclecloud.com
repo3-s3-region=ap-singapore-1
repo3-s3-key=${ORACLE_S3_KEY}
repo3-s3-key-secret=${ORACLE_S3_SECRET}

# Backup schedule (in cron)
# 0 2 * * * pgbackrest --stanza=prod --type=full backup --repo=2
# 0 3 * * * pgbackrest --stanza=prod --type=full backup --repo=3
# 0 4 * * 0 pgbackrest --stanza=prod --type=full backup --repo=1
```

Part 4: Memory and Knowledge Scaling

The Problem with Shared Agent Memory

Agents need memory. Without memory, every conversation starts from zero — the agent cannot recall that you already told it your infrastructure uses Kubernetes, or that the staging environment has known issues with the auth service. This is not just inconvenient; it means agents repeat expensive initialization work on every invocation.

The naive solution is a shared database. Every agent reads and writes to the same memory store. This works until it doesn't — which is around the time you have three agents simultaneously writing conflicting knowledge about the same infrastructure component.

Cognee Across Multiple Servers

Cognee is a knowledge graph system designed for AI agents. It provides structured memory with graph relationships, semantic search, and temporal versioning. Scaling it across multiple servers requires careful attention to the graph consistency model.

```
# Cognee multi-server configuration
import cognee
from cognee.infrastructure.databases.graph import get_graph_config
```

```python
# Each server connects to the same graph backend
# but uses server-local vector cache for performance
async def configure_cognee_for_server(server_id: str):
    await cognee.config.set_graph_db_config({
        "graph_database_provider": "neo4j",
        "graph_database_url": "bolt://db-server:7687",
        "graph_database_username": "neo4j",
        "graph_database_password": os.environ["NEO4J_PASSWORD"],
    })

    await cognee.config.set_vector_db_config({
        "vector_db_provider": "lancedb",
        # Server-local LanceDB for fast vector search
        "vector_db_url": f"/var/lib/cognee/{server_id}/vectors",
    })

    # Namespace isolation: agents on different servers
    # don't pollute each other's knowledge graph
    await cognee.config.set_llm_config({
        "namespace": f"agent-mesh-{server_id}",
    })
```

Critical warning: no multi-tenancy in Cognee (as of March 2026). If you have agents for different customers or projects sharing a Cognee instance, their knowledge will bleed together. Agent A learning that "production DB is at 10.0.1.1" will share that knowledge with Agent B working on a different customer's infrastructure. Run separate Cognee instances per tenant or project, not per agent.

```
# Correct: Isolated Cognee instances per project
   cognee-project-alpha/   (Neo4j db: ai-agents-alpha, LanceDB: /var/lib/cognee/alpha)
   cognee-project-beta/    (Neo4j db: ai-agents-beta,  LanceDB: /var/lib/cognee/beta)
   cognee-project-gamma/   (Neo4j db: ai-agents-gamma, LanceDB: /var/lib/cognee/gamma)

# Incorrect: Shared Cognee instance for all projects
   cognee-shared/          (knowledge bleeds between alpha, beta, gamma)
```

Memory Tiers

Not all agent memory needs the same durability or access speed. Implement tiers:

```
Tier 1 - Ephemeral (in-process, lost on restart)
  Use for: current conversation context, intermediate reasoning steps
  Storage: Python dict / LRU cache
  Access time: <1ms
  Cost: RAM only

Tier 2 - Session (Redis, TTL-based)
  Use for: multi-turn conversation state, task status, user preferences
```

```
  Storage: Redis with TTL (1-24 hours)
  Access time: 1-5ms
  Cost: Redis instance (~$15/month managed)

Tier 3 - Persistent (PostgreSQL)
  Use for: agent decisions, task history, audit trail, billing records
  Storage: PostgreSQL with Patroni
  Access time: 5-50ms
  Cost: Included in database infrastructure

Tier 4 - Knowledge Graph (Cognee/Neo4j)
  Use for: entity relationships, domain knowledge, learned patterns
  Storage: Neo4j + LanceDB (vector index)
  Access time: 20-200ms (semantic search is expensive)
  Cost: $30-100/month depending on graph size
```

Part 5: Cost Scaling

The Exponential Problem

Here is the math that nobody writes about in the "getting started with AI agents" tutorials:

```
Single agent, 100 queries/day:
  Average tokens per query: 2,000 input + 500 output
  Daily tokens: 250,000
  Daily cost (Sonnet 4.6): 250K × $0.003/1K + 50K × $0.015/1K = $1.50/day

Five agents, each 100 queries/day:
  But agents also talk to each other: add 30% inter-agent traffic
  Daily tokens: 5 × 250,000 × 1.3 = 1,625,000
  Daily cost: ~$9.75/day

Ten agents, each 100 queries/day:
  Inter-agent traffic now 60% of total (mesh is busier)
  Daily tokens: 10 × 250,000 × 1.6 = 4,000,000
  Daily cost: ~$24/day

Twenty agents, each 100 queries/day:
  Inter-agent traffic 100%+ of direct traffic
  Daily tokens: 20 × 250,000 × 2.1 = 10,500,000
  Daily cost: ~$63/day
```

The cost does not scale linearly with agent count. It scales super-linearly because agents communicate with each other, and each inter-agent message is a new API call. A mesh of N agents has $O(N^2)$ potential communication paths. Even if only a fraction of those paths are active, the cost profile is distinctly non-linear.

Model Right-sizing

The most impactful cost lever is model selection. Not every task requires Opus.

```
# Model routing by task complexity
def select_model(task: Task) -> str:
    # Simple classification, routing, formatting
    if task.type in ("classify", "route", "format", "extract"):
        return "claude-haiku-4-5"  # $1/$5 per MTok

    # Standard reasoning, code generation, analysis
    elif task.type in ("analyze", "generate", "review", "plan"):
        return "claude-sonnet-4-6"  # $3/$15 per MTok

    # Complex multi-step reasoning, novel problems
    elif task.complexity_score > 0.85:
        return "claude-opus-4-6"  # $5/$25 per MTok

    else:
        return "claude-sonnet-4-6"

# Complexity scoring heuristics
def estimate_complexity(task: Task) -> float:
    score = 0.0
    if len(task.context_tokens) > 50_000: score += 0.2
    if task.requires_multi_step_reasoning: score += 0.3
    if task.involves_ambiguous_requirements: score += 0.2
    if task.has_conflicting_constraints: score += 0.3
    return min(score, 1.0)
```

In practice, after right-sizing, about 60% of my agent queries route to Haiku, 35% to Sonnet, and 5% to Opus. This reduces costs by approximately 70% compared to running everything on Sonnet.

The Cron Job Decision

In the early phases of building the agent mesh, I had 23 automated cron jobs: - Hourly infrastructure health summaries - Daily cost reports - Six-hourly knowledge graph sync - Per-agent heartbeat checks (every 15 minutes) - Nightly log analysis - Weekly trend reports - Continuous alert monitoring with AI-generated summaries

I deleted all of them.

Not gradually. All of them, over one weekend, after an honest accounting of the value they generated versus the cost.

The math was stark. The 23 cron jobs collectively consumed approximately 2.1 million tokens per day — around $12.60/day, or $378/month. The value they generated was: some reports that I usually did not read, health summaries that duplicated what Grafana already showed me, and alert summaries that were less actionable than the raw Alertmanager notifications.

The cron jobs that survived (I eventually brought back four) were the ones where: 1. The AI

generated something I could not get from a conventional monitoring tool 2. The output was regularly read and acted upon 3. The cost per actionable insight was below $1

Everything else was paying to generate text that accumulated in Slack channels and email inboxes without being read. If you are building an agent mesh, audit your scheduled jobs ruthlessly and regularly. Token burn without user value is not automation — it is very expensive log spam.

Part 6: Memory Management at the OS Level

The Swap Problem

Agent development servers run heavy workloads: the AI API client, the web server, the monitoring stack, and often build tooling for the code that agents write. On my Server-2, after any significant build (Gradle projects being the worst offender), memory pressure was severe enough to cause OOM kills of agent processes.

Gradle and Kotlin daemons are particularly aggressive. A typical Gradle build leaves behind daemon processes consuming 3.2GB of RAM — and the daemons are designed to stay alive for fast subsequent builds. On a development server that also runs agents, these daemons compete directly with active agent processes.

```
# Show Gradle daemon memory usage
./gradlew --status
# Output example:
#  PID STATUS   INFO
# 8421 IDLE     7.6.4
#  Memory: heap=512MB, nonHeap=284MB

# Kill all Gradle daemons after build completes
./gradlew --stop

# Or: add to .gradle/gradle.properties to limit daemon lifetime
# (on the server, not in project config)
# org.gradle.daemon.idletimeout=3600000  # 1 hour
# org.gradle.jvmargs=-Xmx2g -XX:MaxMetaspaceSize=512m
```

For swap, I use zram — compressed RAM swap — rather than disk swap. Disk swap on a busy agent server causes catastrophic latency spikes. zram compresses data in RAM using a CPU-efficient algorithm, providing effective swap without the latency penalty of disk I/O:

```
# Install and configure zram (Ubuntu/Debian)
sudo apt install zram-config

# Manual configuration for more control
sudo modprobe zram

# Create a zram device sized to 25% of RAM (good for 32GB servers = 8GB zram)
echo "8G" | sudo tee /sys/block/zram0/disksize
```

```
# Use lz4 algorithm (fast compression, moderate ratio)
echo "lz4" | sudo tee /sys/block/zram0/comp_algorithm

# Format and activate
sudo mkswap /sys/block/zram0
sudo swapon /sys/block/zram0 --priority 100

# Set swappiness low - only swap under real pressure
echo "vm.swappiness=10" | sudo tee -a /etc/sysctl.conf
sudo sysctl -p
```

With this configuration, the server maintains headroom for agent processes even during Gradle builds, and the zram compression ratio (typically 2:1 to 3:1 for agent data) provides effective memory expansion without disk I/O latency.

Summary

Scaling agent systems is fundamentally different from scaling stateless web services because:

1. **Agents are stateful by nature** — their effectiveness depends on accumulated context and memory, which creates synchronization challenges as you scale horizontally.
2. **Costs are super-linear** — every agent you add creates new communication paths with existing agents, and each path is a potential API call. Model right-sizing and ruthless cron job auditing are not optional optimizations; they are survival requirements.
3. **Database HA is non-negotiable, but the failure modes are subtle** — the 22-failover weekend was caused by a configuration mismatch that was invisible under normal load. Implement config consistency checks and run regular failover drills before you need them.
4. **Your Oracle ARM server is reliable, not versatile** — uptime records are not a general endorsement. Match hardware characteristics to workload characteristics.
5. **The right amount of automation is less than you think** — automated AI jobs generate value proportional to their actionability, not their frequency. Delete the cron jobs that nobody reads.

The agent mesh you build at month six will look nothing like the single agent you deployed at month one. That is fine. The architecture should evolve as you learn what the agents are actually doing, what they cost, and what value they return. Scale intentionally, instrument everything, and keep your backup verification schedule more consistent than your deployment schedule.

Next chapter: Lessons Learned — the mistakes I made so you don't have to.

Chapter 16: Lessons Learned — What I Wish I'd Known

> "You think 3 days makes a million-dollar app? " — A user, after I described how long the AI had spent generating code

---

Introduction

Every war story in this book represents a real mistake. Some of them cost money. Some cost sleep. A few cost trust — with users, with stakeholders, or with myself when I looked at a week of work and realized I had built exactly the wrong thing.

This chapter is different from the others. No architecture diagrams. No configuration files for the first half. Just an honest accounting of the ten most consequential mistakes I made building AI agent systems in production, followed by a broader reflection on the human element that the technical literature consistently underserves.

I have organized these roughly in the order a new practitioner will encounter them — which means the first mistakes are operational, the middle ones are architectural, and the last ones are philosophical. You will likely make some of these mistakes anyway, because some lessons only land when they cost you something. But maybe a few of them you can skip.

The Top 10 Mistakes

Mistake #1: Not Setting Budget Alerts (The $500 Lesson)

The first month I ran agents in production without budget alerts, I spent $500 on API calls that I did not intend to make. Not because the agents were doing anything dramatic — no runaway loop, no infinite retry. Just normal operation, multiplied by the fact that I had connected more data sources than I realized, and the agents were processing all of them.

I noticed when I checked my credit card statement. Not when it happened. Three weeks after it happened.

The fix is two lines of configuration. There is no excuse for not doing this on day one.

Anthropic (via AWS Bedrock):

```
# Set a hard spending limit via AWS Budgets
aws budgets create-budget \
  --account-id $AWS_ACCOUNT_ID \
  --budget '{
    "BudgetName": "ai-agents-monthly",
    "BudgetLimit": {"Amount": "200", "Unit": "USD"},
    "TimeUnit": "MONTHLY",
    "BudgetType": "COST",
    "CostFilters": {
      "Service": ["Amazon Bedrock"]
    }
  }' \
  --notifications-with-subscribers '[{
    "Notification": {
      "NotificationType": "ACTUAL",
      "ComparisonOperator": "GREATER_THAN",
      "Threshold": 80,
      "ThresholdType": "PERCENTAGE"
    },
    "Subscribers": [{
      "SubscriptionType": "EMAIL",
      "Address": "you@example.com"
    }]
  }]'
```

Direct Anthropic API — roll your own tracking:

```
# cost-tracker.py - wrap every API call
import anthropic
from dataclasses import dataclass, field
from datetime import datetime, date
import json
import os

PRICES = {
    "claude-opus-4-6":    {"input": 5.00,  "output": 25.00},    # per MTok
    "claude-sonnet-4-6":  {"input": 3.00,  "output": 15.00},
    "claude-haiku-4-5":   {"input": 1.00,  "output": 5.00},
}

DAILY_BUDGET_USD = float(os.environ.get("DAILY_BUDGET_USD", "20.0"))
ALERT_THRESHOLD = 0.80  # Alert at 80% of budget

@dataclass
class CostTracker:
    db_path: str = "/var/lib/agents/costs.json"
    _today_spend: float = field(default=0.0, init=False)
```

```
    def record(self, model: str, input_tokens: int, output_tokens: int) -> float:
        prices = PRICES.get(model, PRICES["claude-sonnet-4-6"])
        cost = (input_tokens / 1_000_000 * prices["input"] +
                output_tokens / 1_000_000 * prices["output"])

        self._today_spend += cost
        self._persist(model, input_tokens, output_tokens, cost)
        self._check_budget()
        return cost

    def _check_budget(self):
        if self._today_spend >= DAILY_BUDGET_USD:
            raise BudgetExceededError(
                f"Daily budget ${DAILY_BUDGET_USD} exceeded. "
                f"Spent: ${self._today_spend:.2f}"
            )
        elif self._today_spend >= DAILY_BUDGET_USD * ALERT_THRESHOLD:
            self._send_alert(
                f"WARNING: {self._today_spend/DAILY_BUDGET_USD*100:.0f}% "
                f"of daily budget used (${self._today_spend:.2f}/${DAILY_BUDGET_USD})"
            )

    def _send_alert(self, message: str):
        # Telegram alert - replace with your preferred channel
        import requests
        requests.post(
            f"https://api.telegram.org/bot{os.environ['TELEGRAM_BOT_TOKEN']}/sendMessage",
            json={"chat_id": os.environ["TELEGRAM_CHAT_ID"], "text": message}
        )

    def _persist(self, model, input_tok, output_tok, cost):
        record = {
            "timestamp": datetime.utcnow().isoformat(),
            "date": date.today().isoformat(),
            "model": model,
            "input_tokens": input_tok,
            "output_tokens": output_tok,
            "cost_usd": cost,
        }
        with open(self.db_path, "a") as f:
            f.write(json.dumps(record) + "\n")

class BudgetExceededError(Exception):
    pass
```

Set the daily budget before you write your first agent. Not after you see the bill.

Mistake #2: Trusting Agents with Infrastructure Config (The CLIProxyAPI Incident)

Early in the project, I gave an agent write access to infrastructure configuration files. The agent's job was to optimize API routing settings. One evening, while I was away from my desk, it decided to "simplify" the CLIProxyAPI routing configuration by removing what it identified as redundant fallback routes.

The fallback routes were not redundant. They were the routes that served 40% of traffic when the primary endpoints rate-limited. Within six minutes of the change, the system was returning errors to 40% of requests. The agent did not notice — it had already moved on to the next task.

The rule: Agents should propose infrastructure changes; humans should apply them. This is not AI skepticism — it is sound change management.

```
# The pattern: generate a change proposal, require human approval
class InfraChangeProposal:
    def __init__(self, agent_id: str, change_type: str, diff: str, rationale: str):
        self.id = str(uuid.uuid4())
        self.agent_id = agent_id
        self.change_type = change_type
        self.diff = diff
        self.rationale = rationale
        self.status = "pending"
        self.created_at = datetime.utcnow()

    def to_approval_message(self) -> str:
        return f"""
INFRASTRUCTURE CHANGE PROPOSAL
ID: {self.id}
Agent: {self.agent_id}
Type: {self.change_type}

Rationale:
{self.rationale}

Diff:
{self.diff}

Reply APPROVE {self.id} or REJECT {self.id}
"""

# Agent generates proposal - does NOT apply it
async def optimize_routing(agent, config_path: str):
    current_config = read_config(config_path)
    proposed_config = await agent.propose_optimization(current_config)

    diff = generate_diff(current_config, proposed_config)
    proposal = InfraChangeProposal(
```

```
        agent_id=agent.id,
        change_type="routing_optimization",
        diff=diff,
        rationale=await agent.explain_changes(diff)
    )

    await send_for_approval(proposal)  # Telegram, Slack, email, whatever
    # Agent stops here. Human applies after review.
    return proposal
```

Infrastructure is the one domain where the cost of an agent mistake is immediate, visible, and potentially customer-impacting. The inconvenience of a manual approval step is worth it.

Mistake #3: Auto-Healing That Costs More Than the Problems It Fixes

Auto-healing sounds ideal: an agent detects a problem, diagnoses it, fixes it, and you wake up to a healthy system. The reality is more complicated.

My first auto-healing agent was triggered by high memory usage on Server-2. Its fix was to restart the heaviest process. That process was a Cognee indexing job that took 45 minutes to restart and reinitialize. The agent triggered the restart, then detected that memory was still high (the restart was in progress), and triggered another restart. Four restarts later, the indexing job had been running for 20 minutes and was almost complete, but the auto-healer had killed it four times at increasingly expensive stages of initialization.

Total cost of the "fix": 3 hours of lost indexing work + 4 × Opus API calls for diagnosis = approximately $8.40 in tokens + ~3 hours of compute.

The original problem: memory was at 87%, which was within normal operating range for that server during indexing. No intervention was needed.

Rules for auto-healing agents: 1. Never restart a process without first checking if it is making progress 2. Implement a minimum cooldown between interventions (I use 30 minutes) 3. Rate-limit interventions per service per day (maximum 2) 4. If the same fix has been applied twice without resolution, stop and alert a human 5. Track the cost of each auto-healing action — if average remediation cost > average incident cost, disable the healer

```
class AutoHealer:
    def __init__(self, cooldown_minutes: int = 30, max_daily_interventions: int = 2):
        self.cooldown = timedelta(minutes=cooldown_minutes)
        self.max_daily = max_daily_interventions
        self._interventions: dict[str, list[datetime]] = defaultdict(list)

    def can_intervene(self, service: str) -> tuple[bool, str]:
        now = datetime.utcnow()
        today_interventions = [
            t for t in self._interventions[service]
            if (now - t).days == 0
        ]
```

```
        if len(today_interventions) >= self.max_daily:
            return False, f"Max daily interventions ({self.max_daily}) reached for {service}"

        if today_interventions:
            last = max(today_interventions)
            if now - last < self.cooldown:
                remaining = self.cooldown - (now - last)
                return False, f"Cooldown active for {service}: {remaining} remaining"

        return True, "ok"

    def record_intervention(self, service: str):
        self._interventions[service].append(datetime.utcnow())
```

Mistake #4: Not Reporting Progress (Agent Silence = User Anxiety)

Agents are slow. A complex task — analyzing a codebase, generating a migration plan, reviewing infrastructure — can take 5 to 15 minutes. During that time, without progress updates, users assume something has gone wrong. They re-submit the task. Now the agent is doing the same job twice. Both completions arrive. Nobody knows which is canonical.

This is a UX problem that compounds into a correctness problem.

```
# Progress reporting wrapper
class ProgressReporter:
    def __init__(self, task_id: str, channel: str):
        self.task_id = task_id
        self.channel = channel
        self._start = time.time()

    async def update(self, step: str, pct_complete: int = None):
        elapsed = int(time.time() - self._start)
        msg = f"[{self.task_id}] {step}"
        if pct_complete is not None:
            msg += f" ({pct_complete}%)"
        msg += f" - {elapsed}s elapsed"
        await self._send(msg)

    async def done(self, summary: str):
        elapsed = int(time.time() - self._start)
        await self._send(f"[{self.task_id}] COMPLETE ({elapsed}s): {summary}")

    async def error(self, message: str):
        elapsed = int(time.time() - self._start)
        await self._send(f"[{self.task_id}] ERROR ({elapsed}s): {message}")
```

```
    async def _send(self, message: str):
        # Implementation depends on your channel (Telegram, Slack, etc.)
        await telegram_send(self.channel, message)

# Usage inside a long-running agent task
async def analyze_codebase(repo_path: str, reporter: ProgressReporter):
    await reporter.update("Scanning repository structure", pct_complete=5)
    files = scan_repo(repo_path)

    await reporter.update(f"Found {len(files)} files, starting analysis", pct_complete=20)
    analysis = await analyze_files(files)

    await reporter.update("Generating recommendations", pct_complete=80)
    recommendations = await generate_recommendations(analysis)

    await reporter.done(f"Analysis complete: {len(recommendations)} recommendations")
    return recommendations
```

The frequency of updates matters: too few and users get anxious; too many and the updates become noise. One update every 30-90 seconds for long tasks is a good baseline.

Mistake #5: Hardcoding Credentials

I have hardcoded credentials in three separate agent projects before learning this lesson. Each time, the credential ended up in a git repository within 48 hours — either committed directly or included in a debug log that was committed. Each time, I rotated the key and spent two hours verifying nothing had been exfiltrated.

The fix is not complicated. Use HashiCorp Vault for secrets that rotate, and environment variables (managed by a secrets manager, not plaintext `.env` files) for everything else.

```
# Vault setup for agent credentials
# Store a secret
vault kv put secret/agents/anthropic \
  api_key="sk-ant-..." \
  rate_limit_tier="tier-3"

# Read in agent process
export ANTHROPIC_API_KEY=$(vault kv get -field=api_key secret/agents/anthropic)

# Or use Vault agent sidecar for automatic renewal:
# /etc/vault-agent/config.hcl
vault {
  address = "https://vault.internal:8200"
}

auto_auth {
```

```
  method {
    type = "approle"
    config = {
      role_id_file_path   = "/etc/vault-agent/role_id"
      secret_id_file_path = "/etc/vault-agent/secret_id"
    }
  }
}

template {
  source      = "/etc/agents/env.tpl"
  destination = "/run/agents/env"
  perms       = "0640"
}
```

The template file (`env.tpl`) generates an environment file that the agent reads at startup. Vault automatically refreshes it when secrets rotate. The agent never needs to know the actual secret value at deployment time.

Mistake #6: Ignoring the Firewall (Server-3 UFW Never Enabled)

Server-3, my database and persistence server, ran for four months with UFW (Uncomplicated Firewall) never enabled. I knew this. I kept meaning to set it up. The server was "behind a private network" and "not directly internet-facing." These are exactly the conditions under which you get breached slowly, by lateral movement, rather than fast, by a direct attack.

Nothing bad happened. I was lucky. Do not rely on luck.

```
# Minimum UFW configuration for a database server
# Run these commands - don't just read them

# Install if not present
sudo apt install ufw

# Default: deny all incoming, allow all outgoing
sudo ufw default deny incoming
sudo ufw default allow outgoing

# Allow SSH from known management IPs only
sudo ufw allow from 10.0.1.0/24 to any port 22

# Allow PostgreSQL only from application servers
sudo ufw allow from 10.0.1.1 to any port 5432  # Server-1 (orchestrator)
sudo ufw allow from 10.0.1.2 to any port 5432  # Server-2 (workload)

# Allow etcd peer communication only between cluster nodes
sudo ufw allow from 10.0.1.1 to any port 2379:2380 proto tcp
```

```
sudo ufw allow from 10.0.1.2 to any port 2379:2380 proto tcp

# Allow Patroni REST API within cluster
sudo ufw allow from 10.0.1.0/24 to any port 8008

# Enable
sudo ufw --force enable
sudo ufw status verbose

# Verify
sudo ss -tlnp | grep -E "5432|2379|2380|8008"
```

Do not wait until you finish the fun parts to enable the firewall. Enable it on day one, before you connect the server to anything important.

Mistake #7: Mixing Agent Data (The Cognee Multi-Tenancy Problem)

This was touched on in Chapter 15, but it deserves elaboration as a mistake. I ran a shared Cognee instance for multiple agents serving different customer projects. Over time, the knowledge graph accumulated facts from all projects.

An agent working on Project Beta asked a question about production database credentials. Cognee retrieved a fact it had learned from Project Alpha — because both projects had entities tagged "production" and "database" and the vector similarity was high enough to surface it.

The agent never exfiltrated the credential. But it used an incorrect database address from the wrong project to construct a migration script, which would have failed loudly at runtime. I caught it in code review. It was a near miss.

Tenant isolation in knowledge systems is not optional when you have multiple customers or projects. Run separate Cognee instances, separate Neo4j databases, or at minimum maintain strict namespace separation with query-time filtering:

```
# Namespace-aware Cognee wrapper
class TenantAwareCognee:
    def __init__(self, tenant_id: str):
        self.tenant_id = tenant_id
        self.namespace_prefix = f"tenant:{tenant_id}:"

    async def remember(self, content: str, entity_type: str, **metadata):
        # Prefix all entities with tenant ID
        await cognee.add(
            content,
            entity_type=f"{self.namespace_prefix}{entity_type}",
            metadata={**metadata, "tenant_id": self.tenant_id}
        )

    async def recall(self, query: str) -> list:
```

```
        # Always filter by tenant
        results = await cognee.search(
            query,
            filters={"tenant_id": self.tenant_id}
        )
        # Double-check: filter out any results without correct tenant
        return [r for r in results if r.metadata.get("tenant_id") == self.tenant_id]
```

Mistake #8: Over-Engineering Monitoring (The v1→v2→v2.2→v2.3 Iteration Trap)

I built a custom agent mesh monitoring system. Then rebuilt it. Then patched it twice. The progression:

- **v1**: Simple Prometheus + Grafana with basic agent metrics. Worked fine. I decided it wasn't comprehensive enough.
- **v2**: Added custom alerting logic, a Python service that correlated agent events, Loki for log aggregation, Tempo for distributed tracing. Took two weeks. The Python service had bugs. The correlations were mostly wrong.
- **v2.2**: Fixed the correlation bugs. Added more dashboards. Added Alertmanager routing for different severity levels. Added a Telegram alert formatter.
- **v2.3**: Realized the Tempo tracing was not actually being used by anyone. Removed it. Simplified Alertmanager routing. Ended up with something that looked a lot like v1 with slightly better alerting.

Total time spent: approximately six weeks across three months.

The monitoring that actually mattered at the end: - Agent heartbeat (is the agent process alive?) - Token cost per agent per day (is this agent within budget?) - Error rate per agent (is the agent succeeding at its tasks?) - Queue depth (is anything backed up?) - Database replication lag (is the replica keeping up?)

That is five metrics. A basic Prometheus setup captures all of them. The elaborate correlation engine was not needed because when agents fail, they fail obviously — either they stop responding, or they start generating errors, or costs spike. You do not need sophisticated correlation to notice any of those things.

Build the simple monitoring first. Add complexity only when you have a specific question that simple monitoring cannot answer.

Mistake #9: Using the Wrong Model for the Wrong Task (Gemini for Infrastructure = Fail)

Not every AI model is equally good at every task. This seems obvious in retrospect. It was not obvious when I was in the middle of a cost-optimization sprint and decided to route infrastructure management tasks to Gemini 2.5 Flash because it was cheaper than Claude Sonnet.

Gemini 2.5 Flash is an excellent model for many tasks. Infrastructure configuration generation is not one of them — at least not for my specific use case of generating Ansible playbooks, Patroni configs, and systemd unit files for a specific stack.

The outputs were syntactically valid but operationally wrong. Ansible tasks that would have executed without errors but not achieved the intended state. systemd unit files with correct syntax but incorrect dependency ordering. Patroni configurations that were plausible but had subtle parameter interactions that caused problems under load.

The issue was not capability — Gemini 2.5 Flash is a capable model. The issue was that my specific stack (Traefik + Patroni + etcd + custom agent IPC) was unusual enough that the model's training distribution did not include much of it. Claude Sonnet, possibly due to different training data or RLHF emphasis, performed noticeably better on this specific combination.

```
# Model selection should be task-type AND domain specific
MODEL_ROUTING = {
    # (task_type, domain): model
    ("generate", "infrastructure"): "claude-sonnet-4-6",    # Better on niche stacks
    ("generate", "python"):         "claude-sonnet-4-6",    # Either works
    ("generate", "javascript"):     "gpt-4o",               # Strong on JS ecosystem
    ("analyze", "logs"):            "gemini-2.5-flash",     # Great at pattern matching
    ("analyze", "metrics"):         "gemini-2.5-flash",     # Fast and cheap
    ("classify", "alerts"):         "claude-haiku-4-5",     # Fast, cheap, reliable
    ("summarize", "documents"):     "gemini-2.5-pro",       # Excellent long-context
    ("reason", "complex"):          "claude-opus-4-6",      # Best for novel problems
}

def route_to_model(task_type: str, domain: str) -> str:
    key = (task_type, domain)
    if key in MODEL_ROUTING:
        return MODEL_ROUTING[key]
    # Default
    return "claude-sonnet-4-6"
```

The lesson is not "use one model for everything" — that is expensive and often wrong. The lesson is: test your specific tasks with your specific models before committing to a routing decision. What works well in general benchmarks may not work well for your particular combination of requirements.

Mistake #10: Not Testing Sub-Agent Outputs (Silent Failures)

When an orchestrator delegates a task to a sub-agent and the sub-agent returns output, the orchestrator typically trusts that output. This is wrong.

Sub-agents fail silently in ways that are not always obvious: - They return plausible-looking but incorrect output - They hallucinate file paths, function names, or configuration values - They correctly identify a problem but propose a fix that addresses a symptom rather than the cause - They return an answer that is correct for a different version of the software

The orchestrator, receiving this output, proceeds as if it is correct. The error propagates downstream. By the time it surfaces — in a failed deployment, a misconfigured service, or a runtime error — the sub-agent's output is three steps removed from the point of failure and hard to trace back.

```
# Output validation framework for sub-agent responses
from typing import Protocol, TypeVar
import jsonschema

T = TypeVar("T")

class OutputValidator(Protocol[T]):
    def validate(self, output: str) -> T:
        """Parse and validate output. Raise ValidationError if invalid."""
        ...

class AnsiblePlaybookValidator:
    REQUIRED_KEYS = {"name", "hosts", "tasks"}

    def validate(self, output: str) -> dict:
        # Step 1: Parse YAML
        try:
            parsed = yaml.safe_load(output)
        except yaml.YAMLError as e:
            raise ValidationError(f"Invalid YAML: {e}")

        # Step 2: Schema validation
        if not isinstance(parsed, list):
            raise ValidationError("Playbook must be a list of plays")

        for play in parsed:
            missing = self.REQUIRED_KEYS - set(play.keys())
            if missing:
                raise ValidationError(f"Play missing required keys: {missing}")

        # Step 3: Dry-run check (ansible-playbook --check)
        result = subprocess.run(
            ["ansible-playbook", "--check", "--syntax-check", "-"],
            input=output.encode(),
            capture_output=True,
            timeout=30
        )
        if result.returncode != 0:
            raise ValidationError(f"Syntax check failed: {result.stderr.decode()}")

        return parsed

# In the orchestrator
```

```
async def delegate_and_validate(sub_agent, task, validator):
    raw_output = await sub_agent.execute(task)

    try:
        validated = validator.validate(raw_output)
        return validated
    except ValidationError as e:
        # Ask sub-agent to fix the output
        fix_request = f"""
Your previous output failed validation:
Error: {e}

Original output:
{raw_output}

Please correct the output and return only the fixed version.
"""
        corrected = await sub_agent.execute(fix_request)
        return validator.validate(corrected)  # Raises if still invalid
```

Build validators for every significant sub-agent output type. The investment pays back immediately the first time a sub-agent returns plausible garbage and your orchestrator rejects it instead of acting on it.

The Human Element

AI Generates Fast; Everything Else Takes 10x Longer

There is a persistent fantasy in the AI tooling space that AI agents will dramatically compress the timeline for building software. This fantasy has a narrow truth in it: AI agents do generate code, plans, and documentation significantly faster than humans working alone.

The fantasy fails because generating code is not the bottleneck.

A user once said to me, after I described how the agent had produced a complete feature implementation in three days: "You think 3 days makes a million-dollar app?" The answer, self-evidently, is no. The three days of AI generation was followed by:

- Two weeks of integration testing (the generated code worked in isolation; integrating it with the existing system required significant rework)
- One week of performance optimization (the generated implementation was correct but naive; it made N+1 database queries and could not handle the production load)
- Three days of security review (the AI had implemented the happy path correctly and completely missed input validation on several endpoints)
- Four days of polish (the UX was functional but not good; real users tried it and found it confusing in ways the AI had not anticipated)

- One week of production monitoring and incident response (four post-deployment issues, three of which the AI had not modeled)

Total actual timeline for "3 days of AI work": approximately 7 weeks.

This is not an argument against using AI agents. The feature might have taken 12-16 weeks without AI assistance. The leverage is real. But the leverage is in the generation phase, which is only one part of software delivery. The build, test, integrate, polish, and operate phases still take approximately as long as they always have.

Plan accordingly. Do not commit to a timeline based on how fast the AI can write code. Commit to a timeline based on how long the entire delivery cycle takes — with AI accelerating the generation phase and not much else.

The Trust Calibration Problem

Working with AI agents daily creates a calibration problem. You see the agent succeed at impressive tasks and start extending trust to domains where the agent is less reliable. You stop checking outputs that the agent usually gets right, which means you stop catching the occasional wrong output in that category.

The opposite also happens: an agent makes a visible mistake, you lose confidence in it, and stop delegating tasks it handles well.

Maintaining accurate calibration — neither over-trusting nor under-trusting — requires systematic output logging and periodic review. I do a weekly spot-check: pick 20 random agent outputs from the week, verify them manually, note any that were wrong. This takes about 45 minutes and keeps my calibration honest.

Predictions for the Field

I am writing this in March 2026. The following reflects where I expect the field to move over the next 12-18 months based on current trajectories. Treat it as informed speculation, not prophecy.

MCP Standardization Will Become Table Stakes

The Model Context Protocol (MCP) has emerged as the dominant standard for connecting AI agents to external tools and data sources. In early 2025, integrating an agent with a new data source meant writing custom code every time. By late 2025, most serious agent frameworks had MCP support. By mid-2026, I expect MCP compliance to be a hard requirement for any enterprise-grade agent integration.

Operationally, this matters because it changes the maintenance burden. Instead of maintaining bespoke integrations with fifteen different data sources, you maintain one MCP adapter and let the frameworks handle the rest.

2M+ Context Windows Will Change Memory Architecture

Context windows have been expanding rapidly: 200K → 1M → and likely 2M+ tokens by late 2026. This will change the economics of the tiered memory architecture described in Chapter 15.

When you can fit an entire codebase, the last six months of incident history, and a complete knowledge graph in a single context window, the case for complex external memory systems weakens. The latency and cost of Cognee graph queries (20-200ms per lookup) becomes harder to justify when you can achieve the same recall by simply stuffing more into context.

The agents that handle this transition well will be the ones built on retrieval abstractions — where the implementation detail of "retrieves from Cognee" vs "retrieves from context" is hidden behind a clean interface that can switch implementations as the cost structure changes.

```python
# Build retrieval abstractions now, optimize implementations later
class AgentMemory(Protocol):
    async def store(self, key: str, content: str, **metadata) -> None: ...
    async def retrieve(self, query: str, limit: int = 5) -> list[MemoryItem]: ...

# Today: Cognee graph retrieval (necessary for 200K context windows)
class CogneeMemory:
    async def retrieve(self, query: str, limit: int = 5) -> list[MemoryItem]:
        return await cognee.search(query, top_k=limit)

# Tomorrow: context-window retrieval (as windows expand to 2M+)
class ContextMemory:
    def __init__(self, context_store: list[str]):
        self._context = context_store

    async def retrieve(self, query: str, limit: int = 5) -> list[MemoryItem]:
        # With 2M context, just include everything relevant
        return self._semantic_filter(self._context, query, limit)
```

Adaptive Reasoning Will Replace Static Prompting

The current dominant approach — writing a detailed system prompt and hoping it covers enough cases — is brittle. Agents fail at edge cases that were not anticipated in the prompt. Adding more instructions to the prompt makes it longer, which degrades performance on the common cases.

The emerging alternative is adaptive reasoning: agents that adjust their approach based on what they observe during task execution, rather than following a static instruction set. Models with stronger reasoning capabilities (Claude Opus 4.6 being the current benchmark, with whatever follows it in 2026-2027) are making this practical.

Operationally, adaptive reasoning changes how you write agent instructions: less specification of exact steps, more specification of goals and constraints. The agent figures out the steps. This is harder to debug when it goes wrong — "the agent did something unexpected" is a more complex failure mode than "the agent did step 7 incorrectly." Plan for more investment in tracing and output logging as adaptive reasoning becomes prevalent.

Final Advice for DevOps Engineers Entering the AI Agent Space

After two-plus years of building, scaling, breaking, and rebuilding AI agent systems in production, here is what I would tell someone starting today:

Start with one agent, one task, one measure of success. The temptation to build a mesh immediately is strong. Resist it. One agent with a clear job and a clear success metric teaches you more about agent behavior than ten agents doing loosely defined work.

Instrument everything from day one. Token counts, costs, latency, error rates, output quality. Not because you will read all the dashboards — you will not — but because when something goes wrong at month six, you will need the historical data to understand when it started.

Treat agent outputs like code from a junior developer. Not with distrust, but with appropriate review. Verify the first few outputs in any new capability domain. Spot-check regularly. Build automated validators for high-stakes output types.

Your DevOps instincts are more transferable than you think. The principles that make reliable infrastructure — redundancy, observability, graceful degradation, change control, least privilege — apply directly to agent systems. The implementation details are different. The principles are the same.

Budget for surprise. Both in money and in time. Agent systems do things you did not predict — sometimes helpfully, sometimes expensively, sometimes both. Keep reserve capacity in your budget and your schedule for the unexpected.

The agents are not the product. The outcomes they enable are the product. An agent that costs $10/day and saves 4 hours of human work per day is good. An agent that costs $10/day and produces outputs that nobody reads is not automation — it is expensive theater. Evaluate agents on the value they create, not on how impressive their outputs look.

The field is moving fast. The specific tools, pricing, and architectures in this book will age. The operational judgment to know when something is working, when it is not, and what to do about it — that ages more slowly. Build that judgment deliberately, from the first agent you deploy.

End of Part 5.

The appendices that follow contain reference material: tool and framework summaries (Appendix A), AI model pricing as of March 2026 (Appendix B), and production checklists you should actually use (Appendix C).

APPENDICES

Appendix A: Tools and Frameworks Reference

> A field guide to the tools that appear throughout this book — what each does, when to reach for it, and what will bite you if you are not careful.

This appendix is organized by category. For each tool, you will find: a plain-English description, the use cases where it shines, and the gotchas that experience surfaces but documentation rarely mentions. Versions and pricing are as of March 2026.

Agent Frameworks

CrewAI

What it does: Orchestrates multiple AI agents in role-based workflows. You define agents with specific roles (researcher, writer, engineer), assign them tasks, and CrewAI manages the execution flow — sequential, parallel, or hierarchical.

When to use it: - Multi-agent pipelines where different agents have distinct expertise areas - Workflows that map naturally to human team structures - Rapid prototyping of agent collaboration patterns

Key gotchas: - Memory between agents in a crew is shared by default — tenant isolation requires explicit configuration - The `process=Process.hierarchical` mode adds a manager LLM call for every task delegation; costs accumulate faster than expected in large crews - Tool definitions are verbose and repetitive; build a tool registry to avoid copy-pasting - Crew output quality degrades significantly when crews exceed 5-6 agents; context becomes too diluted

```
from crewai import Agent, Task, Crew, Process

infra_agent = Agent(
    role="Infrastructure Engineer",
    goal="Analyze and optimize server configurations",
    backstory="Expert in Linux systems, Patroni, and container orchestration",
    tools=[bash_tool, file_reader_tool],
    llm="claude-sonnet-4-6",
    max_iter=5,          # Prevent runaway loops
    memory=True,
```

```
)

analysis_task = Task(
    description="Analyze the PostgreSQL configuration at /etc/postgresql/15/main/postgresql.co
    expected_output="A list of suboptimal settings with recommended values and rationale",
    agent=infra_agent,
)

crew = Crew(
    agents=[infra_agent],
    tasks=[analysis_task],
    process=Process.sequential,
    verbose=True,
)
```

LangChain

What it does: A framework for composing LLM calls, tools, memory, and chains into applications. More of a toolkit than an opinionated framework — gives you primitives to build with rather than a fixed architecture.

When to use it: - Applications that need fine-grained control over LLM interactions - Retrieval-augmented generation (RAG) pipelines - When you need to mix multiple LLM providers in one application

Key gotchas: - The abstraction layer adds meaningful latency (50-200ms per chain step) — measure before assuming it is acceptable - Frequent breaking changes between minor versions; pin versions strictly and test upgrades explicitly - LCEL (LangChain Expression Language) is powerful but produces stack traces that are nearly unreadable when something fails deep in a chain - The LangSmith observability integration is excellent but requires a paid account for production use; budget for it

---

AutoGen (Microsoft)

What it does: Multi-agent conversation framework where agents communicate through structured message passing. Particularly strong at code generation workflows where agents write code, execute it, observe results, and iterate.

When to use it: - Code generation and execution loops - Workflows that benefit from agent-to-agent debate (one agent proposes, another critiques) - Integration with Azure OpenAI (first-class support)

Key gotchas: - Conversation termination requires explicit configuration; without it, agents will continue conversing indefinitely (and billing indefinitely) - The code execution environment runs locally by default; sandbox it before running in production - Message history grows with every exchange — set `max_consecutive_auto_reply` to prevent context window overflow

```
import autogen

config_list = [{"model": "claude-sonnet-4-6", "api_key": os.environ["ANTHROPIC_API_KEY"]}]

assistant = autogen.AssistantAgent(
    name="infra_assistant",
    llm_config={"config_list": config_list},
    system_message="You are an infrastructure engineer. Write and test Ansible playbooks.",
    max_consecutive_auto_reply=5,  # IMPORTANT: always set this
)

user_proxy = autogen.UserProxyAgent(
    name="user_proxy",
    human_input_mode="NEVER",
    code_execution_config={
        "work_dir": "/tmp/autogen-sandbox",
        "use_docker": True,  # Sandbox code execution
    },
    is_termination_msg=lambda x: "TASK_COMPLETE" in x.get("content", ""),
)
```

Claude Code

What it does: Anthropic's AI coding assistant, designed to work within development environments. Reads codebases, writes and edits files, runs commands, and operates through a tool-use loop with human oversight.

When to use it: - Software development tasks: writing, reviewing, refactoring, debugging - Repository exploration and documentation generation - Tasks that require iterative file editing with understanding of the whole codebase

Key gotchas: - Claude Code operates with the file permissions of the user who launched it — run with a dedicated low-privilege user in production-adjacent environments - Long sessions accumulate large context; restart sessions periodically for tasks that span many hours - The tool-use loop can get stuck in "analysis paralysis" on ambiguous tasks; provide explicit success criteria

---

OpenClaw

What it does: A sister AI agent designed to complement Claude Code. Handles tasks Claude Code cannot: web browsing, browser automation, Telegram messaging, scheduled tasks, and device control (camera, screen, location on paired mobile devices).

When to use it: - Web search and live data retrieval - Sending notifications via Telegram, WhatsApp, or Discord - Browser automation (form filling, scraping authenticated pages) - Scheduling reminders and recurring tasks - Coordinating with Claude Code via shared inbox at `~/.openclaw/workspace/`

Key gotchas: - Inter-agent messaging via the shared inbox is file-based; both agents must be running for reliable delivery - Web browsing results include full page content — trim aggressively before including in LLM prompts or costs spike - Mobile device control requires the OpenClaw companion app and explicit user consent per session

```
# Send a task to OpenClaw
openclaw system event --text "Search for Patroni 4.x release notes and summarize breaking chang

# Check inbox for OpenClaw's response
cat ~/.openclaw/workspace/inbox/from-openclaw.md
```

Infrastructure

Docker

What it does: Containerizes applications and their dependencies into portable, reproducible images. The foundation of consistent deployment across development, staging, and production environments.

When to use it: Almost always. If you are not containerizing your agent services, you are creating environment inconsistency problems that will surface at the worst possible moment.

Key gotchas: - Container memory limits interact poorly with Python's GC; set `--memory-swap` equal to `--memory` to prevent swap thrashing - Agent processes that spawn subprocesses (for code execution, tool use) can escape container resource limits via `fork()` unless you use `--pids-limit` - Image layers for Python AI dependencies are large (2-5GB is common); use multi-stage builds to reduce production image size

```
# Multi-stage build for agent service
FROM python:3.12-slim AS builder
WORKDIR /build
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

FROM python:3.12-slim AS runtime
WORKDIR /app
COPY --from=builder /root/.local /root/.local
COPY src/ .
ENV PATH=/root/.local/bin:$PATH
# Never run AI agents as root
RUN useradd -m -u 1000 agent
USER agent
CMD ["python", "-m", "agent.main"]
```

Traefik

What it does: Cloud-native reverse proxy and load balancer with automatic service discovery via Docker labels, Kubernetes, Consul, and etcd. Handles TLS termination, routing, and middleware.

When to use it: - HTTP routing for agent REST APIs - Automatic TLS via Let's Encrypt - Traffic splitting and weighted load balancing across agent instances

Key gotchas: - The dashboard exposes routing configuration by default on port 8080 — restrict access immediately - Middleware order matters; `rateLimit` before `retry` prevents retries from bypassing rate limits - Health check intervals that are too aggressive (< 5s) will generate significant log noise and minor overhead

---

HAProxy

What it does: Battle-tested TCP/HTTP load balancer and proxy. Less feature-rich than Traefik for dynamic environments, but more predictable performance under extreme load and better support for non-HTTP protocols.

When to use it: - PostgreSQL connection pooling front-end (in front of PgBouncer) - TCP load balancing for non-HTTP agent communication - Scenarios where you need sub-millisecond routing decisions with predictable latency

Key gotchas: - HAProxy configuration requires a full restart for most changes (no hot reload for backend changes in older versions) - The stats page should be protected with basic auth at minimum; it exposes connection counts and server health - Session stickiness (`balance source`) can cause uneven load distribution when a small number of IPs generate most traffic

---

Patroni

What it does: High-availability manager for PostgreSQL. Uses a distributed consensus system (etcd, Consul, or ZooKeeper) to manage leader election, automatic failover, and replica promotion.

When to use it: - Any production PostgreSQL deployment that must survive server failures - Multi-server PostgreSQL clusters

Key gotchas: See Chapter 15 for the full postmortem. Short version: - Every node must have identical etcd peer configuration — configuration drift causes spurious failovers - Set `maximum_lag_on_failover` to prevent promoting a significantly-behind replica; 1MB is a reasonable starting point - `use_pg_rewind` must be enabled before a failover, not after — enable it at cluster creation time - Always stop Patroni before stopping etcd; stopping etcd first can trigger an unnecessary failover

---

etcd

What it does: Distributed key-value store providing strong consistency guarantees via the Raft consensus algorithm. Used by Patroni for leader election and cluster state.

When to use it: When you need distributed consensus for leader election, configuration management, or service discovery.

Key gotchas: - Minimum three nodes for meaningful fault tolerance; two nodes provide no protection (both fail simultaneously on any partition) - etcd is sensitive to disk latency — run it on SSDs; HDD-backed etcd will produce election timeouts under load - Enable auto-compaction (`auto-compaction-retention`) or the data directory grows indefinitely and eventually causes OOM - `initial-cluster` value must be identical across all nodes; mismatches cause split-brain symptoms after restarts

---

Monitoring

Prometheus

What it does: Time-series metrics collection and storage. Scrapes metrics endpoints on a configurable interval, stores them in a local TSDB, and provides a query language (PromQL) for analysis and alerting.

When to use it: Always. If you have a production service, it needs Prometheus metrics.

Key gotchas: - Default retention is 15 days; for trend analysis you need either longer retention or a remote storage backend (Thanos, Cortex) - High-cardinality labels (agent IDs, user IDs, request IDs) cause TSDB bloat — keep label cardinality under control - The Prometheus server itself is a single point of failure; run two instances scraping the same targets for monitoring the monitors

```
# Key agent metrics to expose
# In your agent's metrics endpoint:

# Token consumption
agent_tokens_total{model="claude-sonnet-4-6", type="input"} 1523000
agent_tokens_total{model="claude-sonnet-4-6", type="output"} 284000

# Task outcomes
agent_tasks_total{status="success"} 842
agent_tasks_total{status="error"} 17
agent_tasks_total{status="timeout"} 3

# Latency histogram
agent_task_duration_seconds_bucket{le="5"} 623
agent_task_duration_seconds_bucket{le="15"} 801
agent_task_duration_seconds_bucket{le="30"} 839
agent_task_duration_seconds_bucket{le="+Inf"} 862

# Cost tracking
agent_cost_usd_total{agent_id="claude-code-1"} 47.23
```

Grafana

What it does: Visualization and dashboarding platform. Connects to Prometheus, Loki, Tempo, and dozens of other data sources to create operational dashboards and alerts.

When to use it: Alongside Prometheus. The two are effectively a unit.

Key gotchas: - Dashboard configuration stored only in Grafana's SQLite database is not backed up by default — export dashboards as JSON and commit to git - Alert evaluation happens in Grafana but notification routing happens in Alertmanager; understand which system is responsible for what before debugging a missed alert - The default `admin/admin` credentials are genuinely used by people in production; change them on first login

---

Alertmanager

What it does: Handles routing, deduplication, grouping, and silencing of alerts from Prometheus. Sends notifications to Telegram, Slack, PagerDuty, email, and many other channels.

When to use it: Alongside Prometheus for production alerting.

Key gotchas: - Alert routing is a tree structure — an alert matches the first matching route and stops (unless `continue: true`); routing logic bugs cause alerts to go to the wrong channel silently - The `group_wait` and `group_interval` parameters determine how long Alertmanager waits before sending grouped alerts; defaults are often too long for incident response - Silence a specific alert instance (using matchers), not an entire alert name — silencing by name mutes all instances including new ones that fire for different reasons

---

Loki

What it does: Log aggregation system designed to work alongside Prometheus. Uses the same label model as Prometheus for log streams, making log-metric correlation natural in Grafana.

When to use it: - Centralized log storage and search across multiple agent servers - Correlating logs with metrics in the same Grafana dashboard

Key gotchas: - Loki is optimized for log streams with low-cardinality labels — do not put dynamic values (request IDs, agent-generated content) in labels - Log ingestion rate limits will silently drop logs if exceeded; configure `limits_config.ingestion_rate_mb` appropriately for your log volume - The `logql` query language is powerful but the full-text search (`|= "error"`) is a sequential scan — use label selectors to narrow the search space first

---

Tempo

What it does: Distributed tracing backend. Stores and queries traces from OpenTelemetry-instrumented applications.

When to use it: When you have multi-agent workflows where you need to trace a request across multiple agent hops and understand where time is spent.

Key gotchas: - Traces are only useful if every component in the path is instrumented — partial instrumentation produces misleadingly incomplete traces - Storage costs grow proportionally with request volume and trace depth; sample aggressively (1-10% of traces for high-volume paths) - As noted in Chapter 16, Tempo is easy to set up and easy to stop using because nobody looks at traces — define specific questions you will answer with traces before investing in the instrumentation

---

Knowledge and Memory

Cognee

What it does: Knowledge graph memory system for AI agents. Ingests documents, extracts entities and relationships, stores them in a graph database (Neo4j), and provides semantic search via vector embeddings (LanceDB).

When to use it: - Agents that need persistent, structured memory of domain knowledge - Systems where relationships between entities matter (not just document similarity) - Multi-session agent workflows where context must survive restarts

Key gotchas: - No built-in multi-tenancy as of March 2026 — namespace all entities with tenant identifiers manually (see Chapter 16) - Graph construction from large documents is slow and expensive; batch ingestion during off-hours - Semantic search quality degrades if the embedding model used during ingestion differs from the model used during query — keep the embedding model consistent

---

Firecrawl

What it does: Web scraping service optimized for AI ingestion. Crawls websites and returns clean Markdown suitable for LLM context, handling JavaScript rendering, authentication, and rate limiting.

When to use it: - Ingesting web documentation into a knowledge base - Monitoring websites for changes - Research tasks that require current web content

Key gotchas: - API costs accumulate quickly for large crawls; set `maxDepth` and `maxPages` limits explicitly - JavaScript-heavy SPAs may not render correctly without additional configuration; test target sites before committing to a pipeline - Respect `robots.txt` — Firecrawl respects it by default; do not disable this

---

Crawl4AI

What it does: Open-source alternative to Firecrawl. Self-hosted web crawler optimized for AI data extraction, with LLM-based content extraction and async crawling.

When to use it: - When you need on-premises web crawling without third-party API costs - High-volume crawling where Firecrawl costs become prohibitive - Custom extraction logic that Firecrawl's API does not support

Key gotchas: - Requires Playwright for JavaScript rendering — adds significant deployment complexity and memory overhead (~500MB per browser instance) - Async crawling with many concurrent sessions can overwhelm target servers; implement politeness delays - Extraction quality varies significantly by site structure; test and tune extraction prompts per domain

---

Neo4j

What it does: Graph database. Stores data as nodes and relationships with properties, and queries it with Cypher (a graph query language). Used by Cognee as the knowledge graph backend.

When to use it: - When relationship traversal is central to your queries (e.g., "find all services that depend on this database") - Knowledge graphs where entity relationships carry meaning - Social/network analysis on agent interaction patterns

Key gotchas: - Cypher query performance drops dramatically without appropriate indexes — create indexes on frequently-queried node properties at schema creation time - Neo4j's memory configuration (`heap.initial_size`, `pagecache.size`) defaults are too conservative for production; tune based on graph size - The Community Edition is single-instance only; clustering requires Enterprise (paid)

LanceDB

What it does: Embedded vector database built for AI applications. Stores vector embeddings alongside structured data and provides fast ANN (approximate nearest neighbor) search. Used by Cognee for semantic search.

When to use it: - Vector similarity search for semantic memory retrieval - When you want an embedded database (no separate server process) for simplicity - Local-first applications that may eventually need cloud sync

Key gotchas: - LanceDB is embedded — each process opens the database directory directly; concurrent writes from multiple processes require coordination - Index rebuild (for IVF_PQ indexes) is required after significant data additions to maintain search quality; schedule this during low-traffic periods - The `metric` parameter (cosine vs. L2 vs. dot product) must match what the embedding model expects; mismatches produce nonsensical search results

---

Security

HashiCorp Vault

What it does: Secrets management platform. Stores, rotates, and audits access to credentials, API keys, certificates, and other secrets. Supports dynamic secret generation (e.g., temporary database credentials).

When to use it: - Any credential that more than one person or service needs access to - Credentials that should rotate automatically - Audit-required environments where you need to know who

accessed what secret and when

Key gotchas: - Vault's storage backend must be highly available; a Vault outage means agents cannot retrieve secrets and will fail — run Vault in HA mode (Raft integrated storage) - The root token generated at initialization should be used once to configure AppRole auth, then revoked and stored offline - Lease renewal for dynamic secrets must be handled by the client; secrets that expire without renewal cause sudden credential failures

---

UFW (Uncomplicated Firewall)

What it does: Front-end for iptables/nftables on Linux. Simplifies firewall rule management with a human-readable syntax.

When to use it: On every server, from day one. See Chapter 16, Mistake #6.

Key gotchas: - `ufw enable` takes effect immediately — ensure your SSH rule is in place before enabling, or you will lock yourself out - UFW rules persist across reboots by default; rules you add for temporary debugging will survive unless explicitly deleted - `ufw status verbose` shows rules in a readable format; `ufw status numbered` shows them with indices for deletion

---

WireGuard VPN

What it does: Modern, high-performance VPN protocol implemented as a Linux kernel module. Simpler to configure than OpenVPN, faster, and with a smaller attack surface.

When to use it: - Securing communication between agent servers across datacenters - Providing encrypted access channels for administration - Replacing expensive private networking options from cloud providers

Key gotchas: - WireGuard is connectionless — there is no concept of a "connected" peer; use `wg show` and ping to verify connectivity - MTU misconfiguration causes silent packet loss for large payloads; set interface MTU to 1420 as a safe default - Key rotation requires coordinated updates on both peers — build a rotation procedure before you need it

---

AI APIs

Anthropic (Claude)

Models: Opus 4.6, Sonnet 4.6, Haiku 4.5. See Appendix B for pricing.

When to use it: - Complex reasoning and multi-step problem solving (Opus) - General-purpose agent tasks, code generation, analysis (Sonnet) - Classification, routing, formatting, simple extraction (Haiku)

Key gotchas: - Rate limits are per API key, not per model — a spike on Haiku can affect Sonnet availability on the same key - The `beta` header for extended thinking (`interleaved-thinking-2025-05-14`) is required for accessing reasoning traces; plan for this in

your client initialization - Tool use schemas must be valid JSON Schema; invalid schemas produce cryptic errors rather than helpful validation messages - Batch API (50% discount) has a 24-hour processing SLA; not suitable for user-facing latency requirements

---

OpenAI (GPT)

Models: GPT-4o, GPT-4.1, o3, o4-mini. See Appendix B for pricing.

When to use it: - JavaScript/TypeScript ecosystem tasks (strong training data) - Vision tasks alongside text (GPT-4o multimodal) - Complex reasoning with extended thinking budget (o3) - Cost-sensitive reasoning tasks (o4-mini)

Key gotchas: - o3/o4-mini reasoning tokens are billed but not returned in the API response by default; enable `include_reasoning` to see them (increases output costs) - GPT-4.1 has a 1M token context window with no long-context pricing premium — the best price/context-length ratio in the GPT lineup - Organization-level usage limits apply across all API keys in the organization; shared keys can hit org limits unexpectedly

---

Google (Gemini)

Models: Gemini 2.5 Pro, Gemini 2.5 Flash. See Appendix B for pricing.

When to use it: - Long-document processing (up to 1M tokens, with 2M in preview) - Multimodal tasks with video, audio, or images - Log analysis and pattern matching at scale (Flash is fast and cheap) - Tasks where cost is the primary constraint

Key gotchas: - Gemini 2.5 Pro with `thinking_budget` enabled bills thinking tokens at a different rate than output tokens; read the billing documentation carefully - Long-context pricing is 2x for prompts over 200K tokens — this is not prominently displayed; calculate costs before assuming long-context is economical - Vertex AI and Google AI Studio use different API endpoints and authentication; pick one and be consistent

---

DeepSeek

Models: R1, V3.2. See Appendix B for pricing.

When to use it: - Cost-sensitive workloads where quality is acceptable at a fraction of Claude/GPT pricing - Reasoning-heavy tasks where R1's chain-of-thought approach is a good fit - Self-hosted deployments (weights are open)

Key gotchas: - DeepSeek's API endpoint availability from non-Asian regions is occasionally unstable; implement retries and fallback to another provider - The self-hosted version of R1 requires significant hardware (minimum 80GB VRAM for full precision); quantized versions are available but reduce quality - Output formatting with complex tool-use schemas is less reliable than Claude or GPT; validate outputs more aggressively

---

Mistral

Models: Mistral Nemo (and others). See Appendix B for pricing.

When to use it: - European data residency requirements (Mistral is a French company with EU-based infrastructure) - Ultra-low-cost tasks where quality requirements are modest - Self-hosted deployments with consumer GPU hardware (Mistral Nemo runs on 16GB VRAM)

Key gotchas: - Context window is smaller than Claude or GPT offerings; be explicit about context budget in prompts - Mistral Nemo is multilingual but performs significantly better in English and French than other languages

---

Proxy and Load Balancing

CLIProxyAPI

What it does: Routes AI API requests across multiple providers with fallback logic, rate limit handling, and cost tracking. Provides a unified API surface so agent code does not need provider-specific client libraries.

When to use it: - Multi-provider setups where you want automatic fallback when one provider rate-limits - Centralized cost tracking across providers - A/B testing different models on the same task type

Key gotchas: - Configuration changes require careful testing — as noted in Chapter 16, an agent modifying CLIProxyAPI config caused a significant outage. Treat CLIProxyAPI config as infrastructure, not application config - Fallback order matters: if your fallback is a more expensive model, a primary provider outage will silently increase costs - Health check endpoints for each provider backend need to be configured explicitly; without them, a failed provider continues to receive traffic until requests fail

---

Antigravity Manager

What it does: Request scheduling and load balancing layer for AI API calls. Manages request queuing, priority lanes, and budget enforcement across the agent mesh.

When to use it: - Coordinating API usage across multiple agents that share rate limits - Enforcing per-agent or per-task budget caps at the infrastructure level (rather than relying on application code) - Priority-based request scheduling (user-facing requests get higher priority than background jobs)

Key gotchas: - The priority queue can starve low-priority requests during peak load; ensure background jobs have a minimum guaranteed throughput - Integration requires agents to route all API calls through the manager, not directly to providers — architectural change that is hard to retrofit

---

Communication

Telegram Bots

What it does: Delivers notifications, alerts, and interactive commands to Telegram channels or groups via the Telegram Bot API.

When to use it: - Real-time agent status updates and alerts - Interactive agent control (sending commands via Telegram messages) - Cost and budget notifications - Incident alerts with action buttons

Key gotchas: - The Bot API is rate-limited to 30 messages/second to a single chat and 1 message/second to the same group — alert storms will be throttled silently - Markdown formatting in Telegram messages uses a non-standard flavor (`MarkdownV2`); test formatting strings carefully or use `parse_mode=None` and plain text - Bot tokens in environment variables are still credentials; rotate them if exposed, store in Vault

```python
# Minimal Telegram alert sender
import httpx

async def send_telegram_alert(
    bot_token: str,
    chat_id: str,
    message: str,
    parse_mode: str = None
) -> bool:
    url = f"https://api.telegram.org/bot{bot_token}/sendMessage"
    payload = {"chat_id": chat_id, "text": message[:4096]}  # Telegram 4096 char limit
    if parse_mode:
        payload["parse_mode"] = parse_mode

    async with httpx.AsyncClient() as client:
        resp = await client.post(url, json=payload, timeout=10.0)
        return resp.status_code == 200
```

---

tmux-based Inter-Agent Messaging

What it does: Lightweight IPC mechanism using named tmux sessions as message endpoints. One agent writes to a shared file; the target agent's session polls the file or receives a `tmux send-keys` injection.

When to use it: - Local inter-agent communication on the same server - Development and prototyping of agent coordination before investing in a message queue - Fallback communication channel when HTTP-based IPC is unavailable

Key gotchas: - tmux sessions are tied to a specific user; inter-agent messaging across users requires shared file system paths with appropriate permissions - `tmux send-keys` injects keystrokes into the active pane — if the agent session has a different command focused, the keys go to the wrong process - This approach does not scale beyond a single server; if you need cross-server IPC, use

Redis Streams or an equivalent

For current documentation on any tool listed here, use the `docs-seeker` skill or the official project documentation. Versions and features change; the gotchas usually stay the same.

Appendix B: AI Model Pricing Reference (March 2026)

> Prices change. Check provider pricing pages before committing to a budget. This appendix reflects publicly available pricing as of March 1, 2026.

All prices are in USD per **million tokens** (MTok) unless noted. Input = tokens sent to the model (prompt + context). Output = tokens returned by the model (completion).

Quick Reference: Price per MTok

Model	Input	Output	Context	Notes
Claude Opus 4.6	$5.00	$25.00	200K (1M beta)	Best reasoning
Claude Sonnet 4.6	$3.00	$15.00	200K	General purpose
Claude Haiku 4.5	$1.00	$5.00	200K	Fast, cheap
GPT-4o	$2.50	$10.00	128K	Strong multimodal
GPT-4.1	$2.00	$8.00	1M	Best $/context
o3	$10.00	$40.00	200K	Extended thinking
o4-mini	$1.10	$4.40	200K	Cheap reasoning
Gemini 2.5 Pro	$1.25	$10.00	1M (2M preview)	Long context
Gemini 2.5 Flash	$0.30	$2.50	1M	Fastest/cheapest
DeepSeek R1	$0.55	$2.19	128K	Open weights
DeepSeek V3.2	$0.14	$0.28	128K	Ultra-cheap
Llama 4	$0.27	$0.85	128K	Open source
Mistral Nemo	$0.02	$0.02	128K	EU-hosted option

Claude (Anthropic)

Pricing Table

Model	Input ($/MTok$)|$Output$(/MTok)	Context Window	
Opus 4.6	$5.00	$25.00	200K (1M extended beta)
Sonnet 4.6	$3.00	$15.00	200K
Haiku 4.5	$1.00	$5.00	200K

Discounts and Multipliers

Batch API (50% discount): All Claude models support the Batch API for asynchronous processing with a 24-hour SLA.

Model	Batch Input	Batch Output
Opus 4.6	$2.50	$12.50
Sonnet 4.6	$1.50	$7.50
Haiku 4.5	$0.50	$2.50

Long-context pricing (2x multiplier above 200K tokens): When using the 1M token extended context beta on Opus 4.6, tokens beyond the 200K standard window are billed at 2x the base rate.

Tokens in prompt	Opus 4.6 input rate
0 – 200K	$5.00/MTok
200K – 1M	$10.00/MTok

Prompt caching: Anthropic supports prompt caching for repeated system prompts and large context prefixes. - Cache write: 1.25x base input rate - Cache read: 0.1x base input rate (90% savings on cached tokens)

```
# Calculating cost with prompt caching
CACHE_WRITE_MULTIPLIER = 1.25
CACHE_READ_MULTIPLIER = 0.10

def calc_claude_cost(
    model: str,
    input_tokens: int,
    output_tokens: int,
    cached_tokens: int = 0,
    batch: bool = False
) -> float:
    base_prices = {
```

```
        "claude-opus-4-6":   {"input": 5.00,  "output": 25.00},
        "claude-sonnet-4-6": {"input": 3.00,  "output": 15.00},
        "claude-haiku-4-5":  {"input": 1.00,  "output": 5.00},
    }
    prices = base_prices[model]
    discount = 0.5 if batch else 1.0

    new_tokens = input_tokens - cached_tokens
    cost = (
        new_tokens    / 1e6 * prices["input"]  * discount +
        cached_tokens / 1e6 * prices["input"]  * CACHE_READ_MULTIPLIER +
        output_tokens / 1e6 * prices["output"] * discount
    )
    return cost
```

Claude Model Selection Guide

```
Is this a user-facing request requiring fast response?
    Yes → Haiku 4.5 (if simple) or Sonnet 4.6 (if complex)
    No  → Can you batch it?
              Yes → Batch API (50% savings)
              No  → Continue to complexity check

Does it require deep multi-step reasoning or novel problem solving?
    Yes → Opus 4.6
    No  → Sonnet 4.6 for most tasks, Haiku 4.5 for classification/routing
```

OpenAI (GPT)

Pricing Table

Model	Input ($/MTok$)|$Output$(/MTok)	Context	Notes	
GPT-4o	$2.50	$10.00	128K	Multimodal (vision, audio)
GPT-4.1	$2.00	$8.00	1M	No long-context premium
o3	$10.00	$40.00	200K	Extended thinking model
o4-mini	$1.10	$4.40	200K	Cost-efficient reasoning

Discounts and Multipliers

Cached input (50% savings on cache hits):

Model	Standard Input	Cached Input
GPT-4o	$2.50	$1.25
GPT-4.1	$2.00	$1.00

Model	Standard Input	Cached Input
o3	$10.00	$5.00
o4-mini	$1.10	$0.55

o3/o4-mini reasoning tokens: Reasoning tokens are generated internally during extended thinking and are billed at the output token rate. They are not returned in the response by default. Enable `include_reasoning: true` in the API request to receive them (note: this increases billed output tokens).

GPT-4.1 context advantage: GPT-4.1 is the only 1M-context model in the GPT lineup without a long-context pricing premium. For workloads with prompts between 200K and 1M tokens, GPT-4.1 is often the most cost-effective option across all major providers.

OpenAI Batch API

OpenAI's Batch API provides a 50% discount with a 24-hour completion window, matching Anthropic's offering.

Model	Batch Input	Batch Output
GPT-4o	$1.25	$5.00
GPT-4.1	$1.00	$4.00
o4-mini	$0.55	$2.20

Google (Gemini)

Pricing Table

Model	Input 200K ($/MTok)\|Input > 200K($/MTok)	Output ($/MTok)	Context	
Gemini 2.5 Pro	$1.25	$2.50	$10.00	1M (2M preview)
Gemini 2.5 Flash	$0.30	$0.60	$2.50	1M

Discounts and Multipliers

Long-context (2x multiplier above 200K tokens): Unlike GPT-4.1, Gemini applies a 2x multiplier to input tokens exceeding 200K. For very large context windows, calculate carefully.

```
def calc_gemini_cost(
    model: str,
    input_tokens: int,
```

```
    output_tokens: int,
    batch: bool = False
) -> float:
    base_prices = {
        "gemini-2.5-pro":   {"input_short": 1.25, "input_long": 2.50, "output": 10.00},
        "gemini-2.5-flash": {"input_short": 0.30, "input_long": 0.60, "output": 2.50},
    }
    prices = base_prices[model]
    discount = 0.5 if batch else 1.0
    LONG_CONTEXT_THRESHOLD = 200_000

    if input_tokens <= LONG_CONTEXT_THRESHOLD:
        input_cost = input_tokens / 1e6 * prices["input_short"]
    else:
        short_cost = LONG_CONTEXT_THRESHOLD / 1e6 * prices["input_short"]
        long_cost  = (input_tokens - LONG_CONTEXT_THRESHOLD) / 1e6 * prices["input_long"]
        input_cost = short_cost + long_cost

    output_cost = output_tokens / 1e6 * prices["output"]
    return (input_cost + output_cost) * discount
```

Batch API (50% discount): Gemini's batch processing API provides 50% off all token costs with async processing.

Model	Batch Input 200K	Batch Output
Gemini 2.5 Pro	$0.625	$5.00
Gemini 2.5 Flash	$0.15	$1.25

Thinking budget (Gemini 2.5 Pro): Gemini 2.5 Pro supports a configurable thinking budget (0 to 32K thinking tokens). Thinking tokens are billed at the output rate ($10.00/MTok). Disable thinking for simpler tasks to reduce costs significantly.

Free Tier (Google AI Studio)

Gemini 2.5 Flash offers a free tier via Google AI Studio: - 1,500 requests/day free - 1M tokens/minute rate limit (paid tier) - Not suitable for production; use Vertex AI for production deployments

---

Open Source / Self-Hosted

API Pricing (via provider APIs)

Model	Input (/MTok)|Output(/MTok)	Context	Notes	
DeepSeek R1	$0.55	$2.19	128K	Reasoning model, open weights
DeepSeek V3.2	$0.14	$0.28	128K	Best price/performance open model
Llama 4 (Scout)	$0.27	$0.85	128K	Meta open source
Mistral Nemo	$0.02	$0.02	128K	EU-hosted, ultra-cheap

Self-Hosting Costs

When self-hosting open models, API costs are replaced by hardware costs:

DeepSeek R1 (Full, 671B parameters):

```
Hardware required: 8× H100 80GB (or equivalent)
Cloud cost (e.g., Lambda Labs): ~$25/hour for 8× H100
Throughput: ~15-20 tokens/second at full precision
Break-even vs API: ~$25/hr ÷ (15 tok/s × 3600 s/hr) = ~$0.46/1K tokens generated
Self-host is cost-effective above ~10M output tokens/day
```

DeepSeek R1 Distill (8B, Llama-based):

```
Hardware required: 1× RTX 4090 (24GB VRAM)
Cloud cost: ~$0.80/hour
Throughput: ~60-80 tokens/second
Reduced capability vs full R1, but suitable for many tasks
```

Mistral Nemo (12B):

```
Hardware required: 1× RTX 4090 or 2× RTX 3090
Runs on consumer hardware - suitable for on-premises deployment
Very low quality ceiling but sufficient for classification, routing, formatting
```

Monthly Cost Estimates by Agent Load

The following estimates use a realistic token distribution: - Average input tokens per request: 2,500 - Average output tokens per request: 600 - 30-day month

1M Tokens/Day (~400 requests/day)

Provider + Model	Daily Cost	Monthly Cost	Notes
Claude Haiku 4.5	$1.76	$52.80	70% input, 30% output ratio
Claude Sonnet 4.6	$5.25	$157.50	General workload
Claude Opus 4.6	$8.75	$262.50	Full quality
GPT-4o	$4.43	$132.75	
GPT-4.1	$3.55	$106.50	
Gemini 2.5 Flash	$0.53	$15.90	Best value at this scale
DeepSeek V3.2	$0.25	$7.50	Ultra-cheap

Recommended at 1M tokens/day: Mix Gemini 2.5 Flash (bulk tasks) + Claude Sonnet 4.6 (quality tasks). Estimated blended monthly: ~$60-90.

10M Tokens/Day (~4,000 requests/day)

Provider + Model	Daily Cost	Monthly Cost	Notes
Claude Haiku 4.5	$17.60	$528	
Claude Sonnet 4.6	$52.50	$1,575	
Claude Opus 4.6	$87.50	$2,625	Expensive at scale
GPT-4o	$44.25	$1,328	
GPT-4.1	$35.50	$1,065	
Gemini 2.5 Flash	$5.30	$159	
DeepSeek V3.2	$2.52	$75.60	

Recommended at 10M tokens/day: Right-sizing becomes critical. Typical production blended cost with aggressive routing (60% Haiku/Flash, 35% Sonnet, 5% Opus/o3): ~$350-500/month.

100M Tokens/Day (~40,000 requests/day)

At this scale, model right-sizing and the batch API are not optional — they are survival requirements.

Provider + Model	Daily Cost	Monthly Cost	Notes
Claude Sonnet 4.6	$525	$15,750	All Sonnet, no optimization
Claude Sonnet 4.6 (Batch)	$262.50	$7,875	50% batch discount
Gemini 2.5 Flash	$53	$1,590	All Flash
Gemini 2.5 Flash (Batch)	$26.50	$795	
DeepSeek V3.2	$25.20	$756	Lowest API cost
Self-hosted R1 Distill	~$580	~$17,400	Hardware, not tokens

Recommended at 100M tokens/day:

```
Task routing strategy for 100M tokens/day:
  60% → Gemini 2.5 Flash (batch where possible)
        $0.15/MTok batch → ~$2,700/month
  30% → Claude Sonnet 4.6 (batch where possible)
```

```
          $1.50/MTok batch → ~$6,750/month
   8% → DeepSeek V3.2 (cost-sensitive bulk)
          $0.14/MTok → ~$336/month
   2% → Claude Opus 4.6 or o3 (complex reasoning)
          $5.00/MTok → ~$300/month

Estimated total: ~$10,086/month
vs all-Sonnet unoptimized: $47,250/month
Savings: 79%
```

Cost Optimization Techniques

1. Prompt Caching

For agents with long system prompts or repeated context (infrastructure documentation, codebase summaries), prompt caching is the highest-leverage optimization:

```python
# Anthropic cache control headers
response = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=1024,
    system=[
        {
            "type": "text",
            "text": STATIC_SYSTEM_PROMPT,  # Rarely changes
            "cache_control": {"type": "ephemeral"}  # Cache for 5 minutes
        }
    ],
    messages=[{"role": "user", "content": user_message}]
)

# Check cache performance
cache_creation_tokens = response.usage.cache_creation_input_tokens
cache_read_tokens = response.usage.cache_read_input_tokens
# cache_read at 0.1x rate vs cache_creation at 1.25x rate
```

Typical savings: 40-70% on input token costs for agents with large, stable system prompts.

2. Output Token Control

Output tokens cost 3-8x more per token than input tokens across all providers. Constraining output length directly reduces costs.

```python
# Be explicit about output format and length
CONCISE_SUFFIX = """
Respond in JSON only. No explanation. Maximum 200 tokens.
Schema: {"status": "ok|error", "action": "string", "details": "string"}
"""
```

```
# Bad: "Analyze this configuration and tell me what you think"
# Good: f"Classify this config issue severity (critical/high/medium/low). {CONCISE_SUFFIX}"
```

3. Context Window Management

Context that grows unbounded is the most common cause of unexpected cost spikes in long-running agent sessions.

```
class ContextManager:
    def __init__(self, max_tokens: int = 50_000, keep_recent: int = 10):
        self.max_tokens = max_tokens
        self.keep_recent = keep_recent
        self._messages: list[dict] = []

    def add(self, role: str, content: str):
        self._messages.append({"role": role, "content": content})
        self._trim_if_needed()

    def _trim_if_needed(self):
        total = sum(len(m["content"].split()) * 1.3 for m in self._messages)
        if total > self.max_tokens:
            # Keep system message + last N messages
            self._messages = self._messages[:1] + self._messages[-self.keep_recent:]

    @property
    def messages(self) -> list[dict]:
        return self._messages
```

4. Structured Output Schemas

Requesting structured output (JSON mode, tool use schemas) reduces output verbosity and eliminates the tokens spent on conversational framing:

```
Without schema: "Based on my analysis of the provided configuration, I can see that
                 there are several issues that need to be addressed. First, the
                 max_connections setting..." (150 tokens)

With schema:    {"issues": ["max_connections too low", "shared_buffers undersized"],
                 "severity": "high"} (25 tokens)
```

Savings: 60-85% on output tokens for structured data tasks.

Pricing Watch: What Changes and What to Monitor

Prices in this space change more frequently than traditional cloud services. Set a calendar reminder to re-evaluate pricing quarterly. Key things that change:

- **Input/output ratios** — providers periodically adjust the ratio between input and output pricing as they optimize serving infrastructure
- **New model releases** — a new Haiku or Flash-tier model typically offers better capability at similar or lower price; re-evaluate routing decisions after each release
- **Batch API expansion** — not all models supported batch API at launch; check for new additions
- **Free tier modifications** — Google AI Studio free tiers change; do not build production dependencies on free tiers
- **Long-context pricing** — as serving costs decrease, the long-context premium (currently 2x for Gemini, Claude) may decrease or be eliminated

```
# Quick cost check script - run monthly
#!/bin/bash
# cost-check.sh: Pulls last 30 days of token usage from each provider

echo "=== Anthropic ==="
curl -s "https://api.anthropic.com/v1/usage" \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01" | jq '.data[] | {model, tokens: .input_tokens}'

echo "=== OpenAI ==="
curl -s "https://api.openai.com/v1/usage?date=$(date +%Y-%m-%d)" \
  -H "Authorization: Bearer $OPENAI_API_KEY" | jq '.data[] | {model, n_requests}'
```

Pricing data sourced from official provider documentation. Always verify current pricing at provider pricing pages before making budget commitments. Batch API discounts and caching discounts apply to listed base prices.

Appendix C: Production Checklists

> Print these. Laminate them if you are dramatic. The point is to run through them — not to read them once and trust your memory.

Each checklist is designed to be used as-is. Items marked [**BLOCKER**] must be completed before proceeding. Items marked [**RECOMMENDED**] are strongly advised but context-dependent.

---

1. Pre-Deployment Checklist

Run this before every production deployment of a new agent or significant agent change.

Code and Configuration

- ☐ [**BLOCKER**] All secrets are in Vault or environment variables — no hardcoded credentials in code or config files
- ☐ [**BLOCKER**] `git log --all --grep="sk-ant\|api_key\|password\|secret"` returns no results
- ☐ [**BLOCKER**] Agent system prompt reviewed for unintended capability grants (write access, infra changes, credential access)
- ☐ Model selected matches task complexity — not defaulting to Opus for everything
- ☐ Output token limits set on all API calls (`max_tokens` parameter explicit)
- ☐ Context window management implemented — no unbounded message history
- ☐ Tool use schemas validated against JSON Schema spec
- ☐ Error handling covers: API rate limits, timeouts, malformed responses, budget exceeded

Testing

- ☐ [**BLOCKER**] Unit tests pass for all agent tool implementations
- ☐ [**BLOCKER**] Integration test with real API call (not mocked) for the primary agent workflow
- ☐ Edge cases tested: empty input, very long input, adversarial input
- ☐ Sub-agent output validation tested with deliberately malformed responses
- ☐ Budget guard tested — confirm it halts execution when limit is reached
- ☐ Retry logic tested — confirm it does not loop infinitely on persistent errors

Infrastructure

- ☐ [**BLOCKER**] UFW rules updated to allow new agent ports (if applicable)

☐ [**BLOCKER**] Database migrations applied to staging and verified
☐ Docker image built, tagged with git commit SHA, and pushed to registry
☐ Resource limits set in container config (`--memory`, `--cpus`, `--pids-limit`)
☐ Health check endpoint implemented and responding at `/health`
☐ Traefik/HAProxy routing configuration updated and tested in staging
☐ etcd config consistency verified across all cluster nodes (if Patroni cluster involved)

Observability

☐ [**BLOCKER**] Prometheus metrics endpoint exposed and scraping successfully
☐ [**BLOCKER**] Daily budget alert configured (80% threshold notification)
☐ [**BLOCKER**] Hard budget ceiling configured (100% threshold — halt or alert)
☐ Grafana dashboard updated for new agent or new metrics
☐ Alertmanager routing verified — test alert fires to correct channel
☐ Log format is structured JSON (machine-parseable by Loki)
☐ Agent ID included in all log lines and metric labels

Communication

☐ Progress reporting implemented for tasks > 30 seconds
☐ Telegram (or preferred channel) notification tested end-to-end
☐ Error notifications include: agent ID, task ID, error message, timestamp
☐ Deployment notification sent to team channel

Rollback Plan

☐ Previous Docker image tag recorded: `_____`
☐ Rollback command documented and tested in staging
☐ Database rollback procedure documented (if schema changes included)
☐ Rollback decision owner identified: `_____`
☐ Time limit for rollback decision: `_____ minutes after deploy`

---

2. Security Hardening Checklist

Run once at initial server setup, and quarterly thereafter as a review.

Network

☐ [**BLOCKER**] UFW enabled: `sudo ufw status` shows "Status: active"
☐ [**BLOCKER**] Default incoming policy is DENY: `sudo ufw default deny incoming`
☐ [**BLOCKER**] SSH access restricted to known management IP ranges only
☐ [**BLOCKER**] PostgreSQL port (5432) accessible only from application server IPs
☐ [**BLOCKER**] etcd ports (2379, 2380) accessible only within cluster node IPs
☐ All unused ports closed — run `sudo ss -tlnp` and verify each open port
☐ WireGuard VPN configured for cross-datacenter agent communication
☐ Public-facing ports: only 80 (redirect) and 443 (TLS) for web endpoints
☐ Patroni REST API (8008) not exposed to public internet

Authentication and Secrets

- ☐ [**BLOCKER**] Vault initialized and unsealed; root token revoked after initial setup
- ☐ [**BLOCKER**] AppRole auth configured for each agent service
- ☐ All API keys stored in Vault; applications read via Vault agent sidecar
- ☐ Database passwords are randomly generated (min 32 chars), stored in Vault
- ☐ SSH keys: password authentication disabled (`PasswordAuthentication no` in sshd_config)
- ☐ SSH root login disabled (`PermitRootLogin no`)
- ☐ SSH keys rotated within last 12 months
- ☐ Telegram bot tokens stored in Vault (not in .env files)
- ☐ `.env` files are in `.gitignore` — verify with `git check-ignore .env`

System

- ☐ OS packages updated: `sudo apt update && sudo apt upgrade -y`
- ☐ Automatic security updates enabled: `sudo dpkg -l unattended-upgrades`
- ☐ Fail2ban installed and configured for SSH brute-force protection
- ☐ Agent processes run as dedicated low-privilege users (not root, not your user)
- ☐ Docker daemon not accessible by agent user (use rootless Docker or strict socket permissions)
- ☐ `/tmp` mounted with `noexec` option (prevents script execution from temp files)
- ☐ Audit logging enabled: `sudo systemctl status auditd`

TLS and Certificates

- ☐ All HTTPS endpoints use TLS 1.2+ (Traefik default: compliant)
- ☐ Let's Encrypt certificates auto-renewing: check Traefik logs for renewal activity
- ☐ Internal services using self-signed certs have cert pinning or CA validation
- ☐ Certificate expiry monitoring alert configured (alert at 30 days before expiry)

Agent-Specific

- ☐ Agent cannot modify its own system prompt or tool list at runtime
- ☐ Infrastructure change proposals require human approval (no direct write to infra config)
- ☐ Agent output is logged before being acted upon (audit trail)
- ☐ Agent cannot access secrets beyond its assigned Vault policies
- ☐ Inter-agent messages validated — agents do not blindly execute instructions from other agents

---

3. Cost Management Checklist

Run at initial deployment and monthly thereafter.

Budget Controls

- ☐ [**BLOCKER**] Daily budget ceiling configured in `CostTracker` or equivalent
- ☐ [**BLOCKER**] Alert at 80% of daily budget — fires to Telegram or Slack
- ☐ [**BLOCKER**] Hard stop at 100% daily budget — agent halts new requests
- ☐ Monthly budget ceiling set at cloud provider level (AWS Budgets, GCP Billing alerts)
- ☐ Per-agent budget allocation documented: `_____`

☐ Budget review scheduled: first Monday of each month

Model Right-Sizing

☐ Model routing table reviewed and current
☐ Percentage of requests routed to each tier documented:
  – Haiku/Flash tier: _____%
  – Sonnet/GPT-4o tier: _____%
  – Opus/o3 tier: _____%
☐ Average cost per successful task computed: $_____
☐ Tasks routed to Opus that could use Sonnet — reviewed and rerouted if applicable

Token Efficiency

☐ Prompt caching enabled for static system prompts (Anthropic)
☐ Cache hit rate monitored: target > 60% for agents with stable system prompts
☐ Context window trimming implemented — no session exceeds _____ tokens
☐ Output token caps set for all API calls
☐ Structured output schemas in use for data extraction tasks
☐ Verbose/exploratory prompts replaced with targeted, specific prompts

Scheduled Jobs Audit

☐ List all active cron jobs / scheduled agent tasks: `crontab -l && systemctl list-timers`
☐ For each scheduled job, document:
  – Frequency: _____
  – Estimated daily token cost: $_____
  – Last time output was read and acted upon: _____
  – Decision: keep / reduce frequency / delete
☐ Jobs where output is not regularly read: **deleted or disabled**
☐ Total daily spend on scheduled jobs: $_____ (target: < 20% of total daily spend)

Cost Anomaly Detection

☐ Daily cost reviewed each morning (automate with morning Telegram report)
☐ Cost spike threshold defined: alert if daily spend > _____ % of 7-day average
☐ Top 3 cost drivers identified and documented this month:
  1. _____
  2. _____
  3. _____

---

4. Monitoring Setup Checklist

Run when setting up a new server or expanding the monitoring stack.

Prometheus

☐ Prometheus scraping all agent services (verify in `Status > Targets`)

- ☐ Scrape interval set appropriately: 15s for active agents, 60s for batch jobs
- ☐ Retention period configured: minimum 30 days (`--storage.tsdb.retention.time=30d`)
- ☐ Core agent metrics implemented and visible in Prometheus:
  - `agent_tokens_total` (counter, labeled by model and type)
  - `agent_tasks_total` (counter, labeled by status)
  - `agent_task_duration_seconds` (histogram)
  - `agent_cost_usd_total` (counter, labeled by agent_id)
  - `agent_queue_depth` (gauge)
- ☐ Infrastructure metrics scraping:
  - Node Exporter on all servers
  - PostgreSQL Exporter for database metrics
  - Patroni metrics endpoint (`/metrics` on port 8008)
  - etcd metrics endpoint

Alertmanager

- ☐ Alertmanager connected to Prometheus
- ☐ Notification channel configured and tested (Telegram/Slack/PagerDuty)
- ☐ Core alert rules defined:
  - ☐ Agent down (no heartbeat for > 5 minutes)
  - ☐ Daily budget $> 80\%$ consumed
  - ☐ Error rate $> 5\%$ over 15 minutes
  - ☐ Queue depth > 50 for > 10 minutes
  - ☐ Patroni primary not available
  - ☐ Replication lag > 30 seconds
  - ☐ Disk usage $> 80\%$ on any volume
  - ☐ etcd cluster has < 2 healthy members
- ☐ Alert routing verified — test fire each alert and confirm it reaches correct channel
- ☐ Inhibition rules configured — high-severity alerts suppress related low-severity alerts
- ☐ On-call rotation or escalation path documented

Grafana

- ☐ Dashboards for all agent services created
- ☐ Dashboard JSON exported and committed to git: `dashboards/`
- ☐ Core panels per agent dashboard:
  - Token consumption rate (rate over 1h)
  - Cost per day (sum over 24h window)
  - Task success/error rate
  - P50/P95/P99 task duration
  - Queue depth over time
- ☐ Infrastructure dashboard covers:
  - CPU/RAM/Disk per server
  - Network I/O
  - PostgreSQL connections, replication lag, transaction rate
  - etcd leader changes (should be zero in healthy operation)
- ☐ Default `admin` password changed

Logging (Loki)

- ☐ Loki receiving logs from all agent services
- ☐ Log labels are low-cardinality (service, level, agent_id — not request_id or user_id)
- ☐ Log format is structured JSON: `{"level":"error","msg":"...","agent_id":"...","ts":"..."}`
- ☐ Log retention policy configured (30 days default)
- ☐ Log-based alert rule for ERROR/CRITICAL logs with high frequency

---

5. Incident Response Checklist

Use this when something is broken and time is short.

First 5 Minutes: Triage

- ☐ **Identify scope:** Is this one agent, all agents, or infrastructure?
- ☐ **Check Grafana:** Which metrics spiked or dropped at incident start time?
- ☐ **Check Alertmanager:** What alerts are firing? What is the severity?
- ☐ **Check agent heartbeats:** `systemctl status agent-* | grep Active`
- ☐ **Check database:** `patronictl -c /etc/patroni/patroni.yml list` — is there a single primary?
- ☐ **Communicate:** Post incident start to team channel with initial scope assessment

Diagnosis Commands

```
# Is the agent process running?
systemctl status claude-code-agent
journalctl -u claude-code-agent --since "10 minutes ago" -f

# Is the database reachable?
psql -h localhost -U postgres -c "SELECT version();"
patronictl -c /etc/patroni/patroni.yml list

# Is etcd healthy?
etcdctl --endpoints=http://server1:2379,http://server2:2379,http://server3:2379 \
  endpoint health

# What is consuming memory?
free -h
ps aux --sort=-%mem | head -20

# What is the current API error rate?
# (Prometheus query)
rate(agent_tasks_total{status="error"}[5m]) /
rate(agent_tasks_total[5m])

# What is today's spend so far?
```

```
tail -n 1000 /var/lib/agents/costs.json | \
  python3 -c "import sys,json; data=[json.loads(l) for l in sys.stdin]; \
  print(f'Today spend: \${sum(d[\"cost_usd\"] for d in data):.2f}')"
```

Containment

- ☐ If cost runaway: disable scheduled jobs first (`systemctl stop agent-scheduler`)
- ☐ If agent loop: `systemctl stop <agent-service>`; do not restart until root cause is known
- ☐ If database failover storm: stop Patroni on all nodes before restarting etcd
- ☐ If security incident: isolate affected server from network before investigation: `sudo ufw deny incoming`
- ☐ If API provider incident: check provider status page before extensive internal debugging

Resolution

- ☐ Root cause documented in writing (even one sentence)
- ☐ Fix applied and tested in staging if time allows; applied directly in production if critical
- ☐ Agent restarted: `systemctl start <agent-service>`
- ☐ Verify recovery: check Grafana dashboard for normalization over 10 minutes
- ☐ Budget check: confirm spend did not spike beyond acceptable range during incident
- ☐ Communicate resolution to team channel with duration and root cause summary

Post-Incident (within 48 hours)

- ☐ Written postmortem: what happened, timeline, root cause, fix, prevention
- ☐ Monitoring gap identified (if the incident was not caught by existing alerts, add an alert)
- ☐ Runbook updated with any new diagnostic commands discovered during incident
- ☐ Checklist updated if a step was missing

---

6. Agent Debugging Checklist

Use when an agent is producing incorrect, inconsistent, or unexpected outputs.

Confirm the Problem

- ☐ Can you reproduce the problem consistently? If not, it may be model non-determinism — set `temperature: 0` and retry
- ☐ Is the problem in the agent's reasoning, its tool calls, or its final output?
- ☐ Is the problem new (regression) or has it always behaved this way?
- ☐ Does the problem occur with all inputs or specific inputs?

Inspect the Agent

```
# Enable verbose logging for a debug session
import anthropic
import json
```

```
client = anthropic.Anthropic()

# Log the full request
request = {
    "model": "claude-sonnet-4-6",
    "max_tokens": 1024,
    "system": agent.system_prompt,
    "messages": agent.context.messages,
    "tools": agent.tools,
}
print("REQUEST:", json.dumps(request, indent=2))

response = client.messages.create(**request)
print("RESPONSE:", response.model_dump_json(indent=2))
print("STOP REASON:", response.stop_reason)
print("TOOL CALLS:", [b.name for b in response.content if b.type == "tool_use"])
```

Common Failure Patterns

Symptom	Likely Cause	Fix
Agent refuses task it previously completed	System prompt conflict or new safety update	Review system prompt for ambiguous instructions
Tool called with wrong arguments	Schema mismatch or unclear tool description	Improve tool description; add examples to schema
Agent loops without completing	No clear termination condition	Add explicit success/failure criteria to system prompt
Inconsistent outputs on same input	`temperature > 0`	Set `temperature: 0` for deterministic tasks
Agent ignores tool results	Tool result format unexpected	Log raw tool results; verify format matches what agent expects
Context window exceeded	Unbounded message history	Implement `ContextManager` with explicit trim policy

Symptom	Likely Cause	Fix
High rate of "I cannot help with that"	System prompt too restrictive	Review and relax unnecessary restrictions
Sub-agent returns plausible but wrong output	No output validation	Add validator for this output type

Prompt Debugging

```python
# Minimal reproducible example - strip everything non-essential
debug_response = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=512,
    # Start with NO system prompt - add it back to isolate if it causes the issue
    messages=[
        {"role": "user", "content": PROBLEMATIC_INPUT}
    ]
)
# If it works without system prompt, the issue is in the system prompt
# Add system prompt back, then tools, then context - binary search the issue
```

Tool Use Debugging

- ☐ Tool schema is valid JSON Schema (validate at `jsonschema.org`)
- ☐ Tool description explains when NOT to use the tool (overuse is common)
- ☐ Tool returns are formatted as the agent expects (string vs object)
- ☐ Tool errors return useful error messages, not Python tracebacks
- ☐ Tool execution is logged with input and output for every call

---

7. Model Selection Decision Tree

Use when choosing which model to call for a new task type.

```
START: What is the primary task?

  Classification / Routing / Tagging
      Claude Haiku 4.5 or Gemini 2.5 Flash
        Low complexity, high volume → Flash (cheaper)
        Need strong reasoning → Haiku

  Data Extraction (from structured documents)
      Gemini 2.5 Flash (fast, cheap, good at extraction)
        If document > 200K tokens → Gemini 2.5 Pro (long context)

  Code Generation
```

```
      Python, infrastructure, DevOps → Claude Sonnet 4.6
      JavaScript/TypeScript ecosystem → GPT-4o or GPT-4.1
      Simple boilerplate → Claude Haiku 4.5
      Novel architecture design → Claude Opus 4.6

  Code Review
      Claude Sonnet 4.6 (reliable, cost-effective)
        Critical security review → Claude Opus 4.6

  Document Summarization
      < 50K tokens → Claude Sonnet 4.6 or Gemini 2.5 Flash
      50K-200K tokens → Gemini 2.5 Pro or GPT-4.1
      > 200K tokens → Gemini 2.5 Pro (cheapest long-context)

  Reasoning / Problem Solving
      Clear problem, known domain → Claude Sonnet 4.6
      Ambiguous problem, novel domain → Claude Opus 4.6
      Math / logic / formal reasoning → o3 or o4-mini

  Infrastructure Config Generation
      Claude Sonnet 4.6 (best on niche stacks from book experience)
        Do NOT use Gemini Flash for infra config (see Chapter 16, Mistake #9)

  Alert / Log Analysis (bulk)
      Gemini 2.5 Flash (fast pattern matching, low cost)
        Batch API if not time-sensitive (50% additional savings)

  Multimodal (images, video, audio)
       Image + text → GPT-4o or Gemini 2.5 Pro
       Video analysis → Gemini 2.5 Pro (native video support)
       Audio transcription + analysis → Gemini 2.5 Pro

SECONDARY QUESTION: Is this latency-sensitive?
  Yes → Avoid batch API; prefer Flash/Haiku tier for non-critical tasks
  No  → Consider batch API (50% savings, 24h SLA)

TERTIARY QUESTION: Is this a recurring/scheduled task?
  Yes → Apply stricter cost scrutiny (see Cost Management Checklist)
        If value is not clear → delete the job (Chapter 16, Mistake #8)
  No  → Proceed with model selection above
```

Checklist Maintenance

These checklists are living documents. After every significant incident or deployment, review and update the relevant checklist:

- Add any diagnostic step you had to figure out on the fly
- Remove steps that proved irrelevant for your specific stack
- Adjust thresholds to match your operational reality (budget ceilings, alert thresholds)
- Date-stamp major revisions

Store the current version of these checklists in your git repository alongside your infrastructure code. Run `git log appendices/C-checklists.md` to see revision history.

```
# Quick checklist health check - verify key guards are in place
check_production_guards() {
  echo "=== Budget Guard ==="
  grep -r "BudgetExceeded\|daily_budget\|DAILY_BUDGET" src/ | wc -l
  echo "  (should be > 0)"

  echo "=== Secret Scan ==="
  git log --all -S "sk-ant\|sk-proj\|AIza" --oneline | wc -l
  echo "  (must be 0)"

  echo "=== UFW Status ==="
  sudo ufw status | head -1

  echo "=== Patroni Health ==="
  patronictl -c /etc/patroni/patroni.yml list 2>/dev/null | grep -c "Leader"
  echo "  (should be 1)"

  echo "=== Active Cron Jobs ==="
  crontab -l 2>/dev/null | grep -v "^#" | grep -c "."
  echo "  (review each one)"
}
```

End of Appendix C. End of book.

AI Agents in Production: War Stories from a DevOps Engineer Second Edition, March 2026 Written with the help of Claude Opus 4.6 — the very tool this book documents.