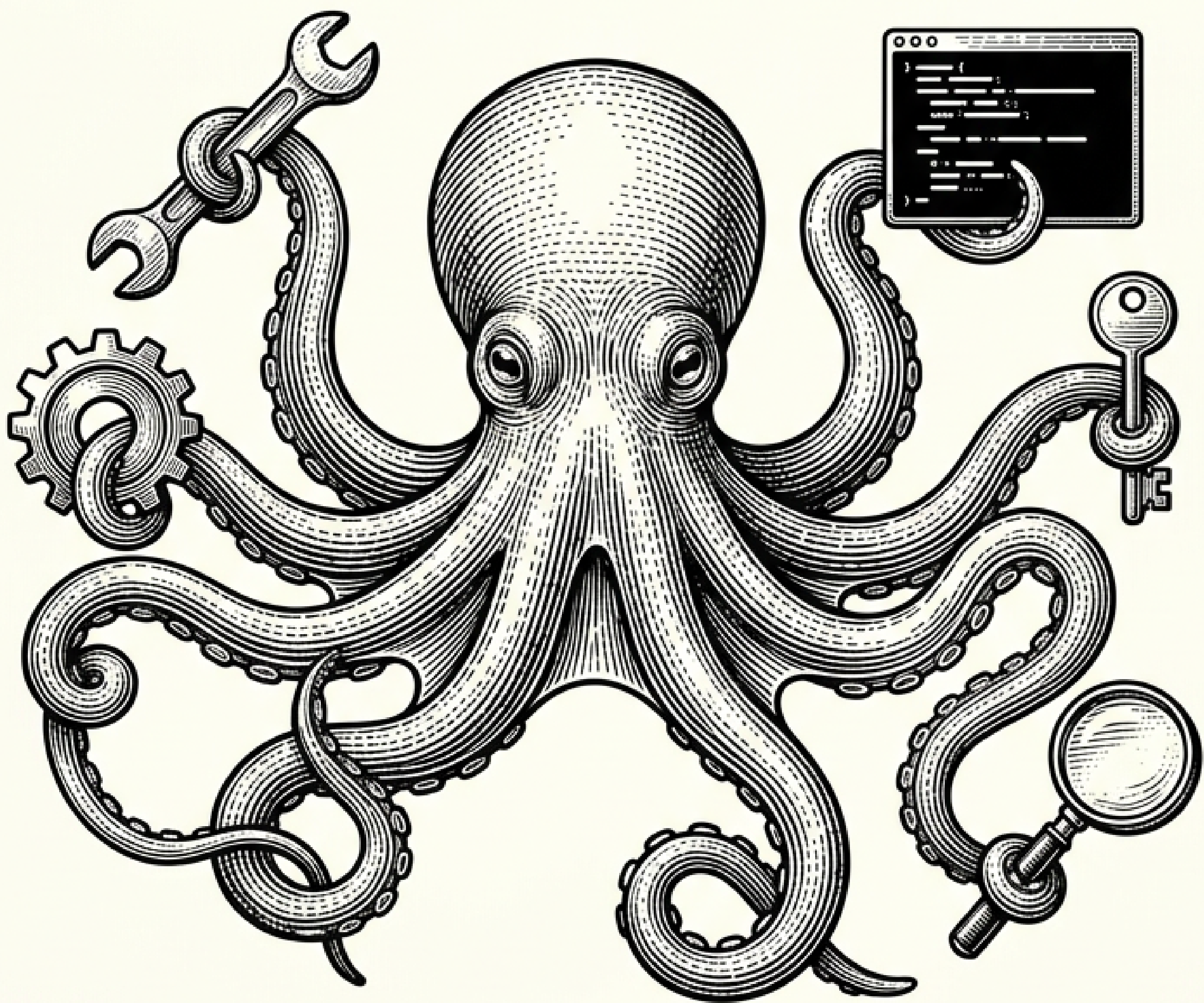
AI Agents in Production

War Stories from a DevOps Engineer



Second Edition

Do Cao Hieu

- AI Agents in Production: War Stories from a DevOps Engineer
  - Table of Contents
    - Part 1: Foundations
    - Part 2: Architecture
    - Part 3: Operations
    - Part 4: Advanced Topics
    - Part 5: The Future
    - Appendices
- PART 1: FOUNDATIONS
- Chapter 1: Why This Book Exists
  - The Email I Never Expected to Write
  - The Gap Between the Demo and Reality
  - Who Is Writing This
  - What Changed From 2024 to 2026
    - The Context Window Revolution
  - The Foundational War Story: Over a Dozen Accounts Suspended in One Day
    - Background
    - What Happened
    - The Cascade
    - What Changed After
  - What This Book Covers
    - What This Book Does Not Cover
  - Who This Book Is For
  - A Note on Honesty
  - How to Use This Book
  - The Setup That Makes This All Real
  - Before We Begin
  - Key Takeaways
- Chapter 2: What Are AI Agents, Really?
  - The Definition Problem
  - What a Language Model Actually Is
  - The Agent Loop
  - What Distinguishes an Agent from a Chatbot
  - Types of Agent Architecture
    - Single Agent
    - Multi-Agent (Orchestrator + Specialists)
    - Mesh / Swarm
  - The Gateway Agent Architecture: A Real Multi-Agent System

- Tools: The Agent’s Hands
    - Tool Design Principles
- War Story: The Agent Mesh Monitor That Burned Itself Down
    - The Setup
    - The Problem
    - The Worse Problem
    - The Decision
    - The Lesson
- Agent vs. Chatbot vs. Copilot: The Practical Table
- The Node.js Version: For the Server-Side Engineers
- Termination: The Most Important Thing You Are Not Thinking About
- Key Takeaways

- Chapter 3: Choosing Your Model: A DevOps Engineer’s Guide
    - The Model Selection Problem
    - The Pricing Landscape (March 2026)
        - Claude 4.x — Anthropic
        - GPT Series — OpenAI
        - Gemini — Google
        - Open Source / Self-Hosted / Third-Party
    - How to Think About Model Selection
        - The Capability Ladder
        - The Selection Matrix
    - War Story: Routing Infrastructure Tasks to the Wrong Model
        - The Setup
        - What Happened
        - The Lesson
    - War Story: The API Proxy Round-Robin with 15 Provider Accounts
        - The Architecture
        - The Day All 15 Accounts Hit 100% Simultaneously
    - War Story: The Model Mismatch Bug (HTTP 400 INVALID_ARGUMENT)
        - The Setup
        - What Happened
        - The Larger Lesson
    - Building a Model Selection Policy
    - The Open Source Reality Check
    - Monitoring Your Model Selection Over Time
    - Quick Reference: Selection Rules
    - Key Takeaways
- PART 2: ARCHITECTURE

- Chapter 4: Agent Architecture Patterns
  - Introduction
  - Pattern 1: Single Agent (Claude Code Standalone)
    - When It Works
    - The Real Constraint: Context Window
    - Single Agent Anti-Patterns
  - Pattern 2: Multi-Agent Orchestration
    - How It Works in Practice
    - Sub-Agent Context Budget
    - The Orchestration Pattern I Use
    - War Story: The 15-Agent Pileup
    - The Cleanup Problem
  - Pattern 3: Agent Mesh
    - Evolution: v1 → v2.3
    - The Full Mesh Monitor Script
    - Deployment: systemd Timer Instead of Cron
    - What the Mesh Can't Fix
  - Pattern 4: Event-Driven Agents
    - The Two-Layer Alert Architecture
    - Why Two Layers?
  - Pattern 5: Hybrid Architectures
    - Model Selection by Task
  - Architecture Decision Guide
    - The Coordination Tax
    - Failure Mode Catalog
    - Observability Across Architectures
  - Key Takeaways
- Chapter 5: Context Engineering — The Art of Token Optimization
  - Introduction
  - What Is Context Engineering?
  - Token Budget Anatomy
  - The Crisis: Provider Accounts Suspended
    - The Context File Audit
  - The Three-Tier Context Architecture
    - Tier 1: Always-Injected (System Prompt)
    - Tier 2: On-Demand Reference (Not Injected)
    - Tier 3: Archived / Deleted
  - CLAUDE.md and AGENTS.md as Context Engineering Tools
    - What CLAUDE.md Should Contain
    - The Instruction-to-Pointer Ratio

    - Heartbeat Optimization: 3 Min → 30 Min
        - What the Heartbeat Actually Needs
    - The Orphan Session Problem
        - How to Find Orphaned Sessions
        - Prevention
    - Cross-Session Context: Why Absolute Paths Are Non-Negotiable
    - The Super-Prompt Failure: Why Large Tasks Need Phase Splits
        - The Phase Split Solution
    - Skills Folder Hygiene: The 145 → 105 Cleanup
    - Context Budget Calculator
    - Token Counting Utilities
    - Practical Optimization Checklist
    - The Mindset Shift
    - Key Takeaways
- Chapter 6: Tool Use and Function Calling
    - Introduction
    - How Tool Use Works: The Request-Response Cycle
    - Claude `tool_use` vs OpenAI / Gemini `function_calling`
        - Claude (Anthropic)
        - OpenAI
        - Gemini (Google)
        - Platform Comparison Summary
    - Designing Good Tool Schemas
        - The Four Properties of a Good Tool Schema
        - A Production-Grade Tool Implementation
    - Tool Use with Interleaved Thinking (Claude Opus 4.6)
    - MCP: Model Context Protocol
        - How MCP Works
        - Setting Up MCP in Claude Code
        - Real Example: Cognee MCP Integration
        - Deploying the Cognee MCP Server
        - Writing Your Own MCP Server
    - Parallel Tool Execution
    - War Story: The Agent That Cut Its Own Feet Off
        - The Setup
        - What Happened
        - The Lesson: Guardrails Must Be Explicit
    - API Key Management: The Proxy Key Typo That Cost Hours
    - Tool Use Patterns for Production
        - Pattern: Read-Before-Write

        - Pattern: Dry-Run First
        - Pattern: Idempotent Tools
    - Key Takeaways
- PART 3: OPERATIONS
- Chapter 7: Deploying AI Agents to Production
    - The Server Topology
    - Process Manager Reality Check: Docker vs Systemd vs PM2
        - The Service Inventory (Server-1 Reality Check)
        - When to Use Docker
        - When to Use Systemd
        - When to Use PM2 (And When Not To)
    - Traefik with docker-socket-proxy (The Right Way)
    - VPN Binding for Sensitive Ports
    - HAProxy for Database Connection Pooling
    - War Story #1: The Firecrawl Network Isolation Bug
        - What Happened
        - Root Cause
        - The Fix
        - Lesson Learned
    - War Story #2: The Server That Had No Firewall
        - What Happened
        - The Damage Assessment
        - The Safe Hardening Procedure
        - What We Added to the Audit Checklist
    - Docker Compose Patterns for Agent Services
        - Pattern 1: Agent with Isolated Network
        - Pattern 2: Multi-Container Agent Stack
        - Pattern 3: Systemd Unit for Agent Processes
    - Secrets Management
    - Health Checks and Dependency Management
    - Deployment Checklist
    - Summary
- Chapter 8: Cost Management — Don't Go Bankrupt Running AI Agents
    - The $500 Gemini Incident
        - What Happened
        - What Should Have Happened
        - Getting the Refund
    - Budget Alerts Are Not Optional
        - Google Cloud Budget Alerts

    - Anthropic Claude Budgets
  - Trial Credits and the Models That Don’t Count
  - The Token Optimization Crisis
    - The Problem: Baseline Token Consumption
    - The Optimization
    - Heartbeat Interval Optimization
    - Deleting All Auto Cron Jobs
  - Orphan tmux Sessions: The Silent Quota Killer
  - Local API Proxy: Free Quota Through Account Rotation
    - How It Works
    - The Quota Reality
    - Latency Trade-off
  - Model Tiering Strategy
    - Batch API: 50% Discount
  - Cost Monitoring Dashboard
  - The Cost of “Just Checking”
  - Budget Monitoring Script
  - Key Cost Management Rules
- Chapter 9: Monitoring AI Agents in Production
  - What You Actually Need to Monitor
  - The Monitoring Stack
    - Prometheus Configuration
    - Alert Rules for Agent Services
    - Alertmanager Configuration
  - Docker Events Listener: Real-Time Container Alerts
  - Agent Mesh Monitor: Cross-Server Health Checking
    - Per-Server Health Check Scripts
  - War Story #1: The Wrong Service Check
    - What Happened
    - The Fix
    - Lesson Learned
  - War Story #2: The 102-Restart Crash Loop
    - What Happened
    - Root Cause: Two Processes Running
    - How It Happened
    - The Fix
    - Lesson Learned
  - Neural Memory Audit: When Agents Have Their Own Health
    - Memory Health Metrics to Track

- Grafana Dashboard Structure
  - Dashboard 1: Infrastructure Overview
  - Dashboard 2: AI Agent Metrics
  - Dashboard 3: Logs (Loki)
  - Dashboard 4: Alerts History
- Log Aggregation with Loki
- The Complete Monitoring Checklist
- Summary

- Chapter 10: Debugging AI Agents — When Things Go Wrong
  - The Debugging Mindset Shift
    - Traditional debugging assumptions that don't hold for agents:
    - The Agent Debugging Stack
  - Failure Mode 1: Agent Modifying Its Own Infrastructure
    - The Incident
    - Why It Happens
    - Prevention
    - Recovery
  - Failure Mode 2: API Key Typos and Credential Errors
    - The Incident
    - Debugging Authentication Issues
    - The Pattern: Credential Inconsistency Across Files
  - Failure Mode 3: Permission Errors
    - Common Permission Errors in Agent Contexts
    - Building a Permission Diagnostic Tool
  - Failure Mode 4: Rate Limits
    - The Vertex AI 429 Problem
    - Rate Limit Handling Patterns
    - API Proxy Auto-Switch on Quota Exhaustion
  - Failure Mode 5: Agent Not Reporting Progress
    - The Problem
    - The Pattern That Prevents This
  - Failure Mode 6: Context Limit Exceeded Mid-Task
    - The Problem
    - Prevention: Context Budget Management
    - The Chunking Pattern
  - Failure Mode 7: Model Mismatch
    - The Incident
    - Detecting Model Mismatches
  - War Story: SSR CSS Hash Mismatch
    - What Happened
    - The Fix

        - The Lesson
    - War Story: Config Sync Across Three Servers
        - What Happened
        - The Diagnosis
        - Prevention: Config Sync Script
        - The Better Long-Term Solution: Centralized Config
    - War Story: The Etcd Split-Brain
        - What Happened
        - The Fix
    - War Story: The Reasoning Parameter Bug
        - What Happened
        - The Fix
    - War Story: Claude Code Stuck at Trust Dialog
        - What Happened
        - The Fix (Short-term)
        - The Fix (Long-term)
    - War Story: Sub-Agent Silent Failure
        - What Happened
        - What Actually Happened
        - Prevention: Verification Steps in Sub-Agent Tasks
    - The Debugging Checklist
    - Summary
- PART 4: ADVANCED TOPICS
- Chapter 11: Security Hardening for AI Agent Infrastructure
    - The Day the AI Became Our Biggest Security Hole
    - The Threat Model Has Changed
    - Part 1: Vault Token Management
        - The Root Token Problem
        - Token Rotation Automation
        - Vault Seal/Unseal Procedures
    - Part 2: Network Hardening — Ports That Should Never Be Public
        - The Discovery
        - Enabling UFW From Scratch (Without Locking Yourself Out)
        - Binding Sensitive Services to VPN Interface
        - Cloud ARM Server: NFS/rpcbind Cleanup
    - Part 3: SSH Hardening
        - PermitRootLogin and PasswordAuthentication
        - SSH Key Rotation When Onboarding New Machines
        - Service Restart After Password Rotation

  - Part 4: Cleartext Credentials in Configuration Files
    - Patroni.yml — Passwords in Plain Sight
    - Hardcoded Credentials in Application Code
  - Part 5: Docker Security — The docker.sock Problem
    - Traefik and docker.sock
  - Part 6: HAProxy Stats — The Default Password Nobody Changed
  - Part 7: Vault Policies for Agent Access
  - Part 8: The AI Agent Security Checklist
    - Pre-Deployment
    - Infrastructure Baseline
    - Operational
  - Lessons Learned
- Chapter 12: AI Image Generation at Scale
  - The Batch Scale Problem
  - The Architecture: Batch Processing at Scale
    - Starting Point: What We Had
  - Hitting Quota Mid-Batch
    - The Error
    - Option 1: Request a Quota Increase (Takes Days)
    - Option 2: Hot-Swap API Keys Across Projects
  - The Delimiter Disaster
    - How the Bug Appeared
    - The Fix: Auto-Detect and Switch to TAB Delimiter
  - Checkpoint and Resume: Never Start Over
    - The Principle
    - Resuming After a Failure
  - The Logo Experiment: When AI Isn’t the Right Tool
    - The Pitch
  - Rate Limits: The textembedding-gecko Bottleneck
    - The Situation
    - Requesting a Quota Increase
  - Veo 3: Video Generation
    - The Experiment
    - What Actually Happened
  - The POD Skill: Five Script Versions
  - Lessons Learned
- Chapter 13: Knowledge Management: RAG, Graphs, and Memory
  - Why RAG Alone Isn’t Enough

- Architecture Overview
  - The Stack
  - Why Server-3?
- Cognee Setup
  - Installation and Configuration
  - Docker Compose for Neo4j
- Dataset Organization
  - The Four Datasets
  - Why Separate Datasets Matter
- Importing 400 DevOps Books
  - The Import Pipeline
  - War Story: The 409 Conflict Storm
  - War Story: Vertex AI Rate Limits Stall Everything
- The MCP Server: Agent-Accessible Knowledge
  - MCP Configuration
  - Claude Code MCP Configuration
- Crawl4AI vs Firecrawl: Choosing Your Web Ingestion Tool
  - Firecrawl
  - Crawl4AI
  - Decision Matrix
- Neural Memory: What the Graph Looks Like
- Lessons Learned

- Chapter 14: Multi-Agent Orchestration in Practice
  - The Problem with One Agent
  - Architecture: Gateway Agent Pattern
    - The Core Design
    - Shared Workspace Communication
  - Tmux-Based Communication: Sending Keystrokes Between Sessions
    - How ping-claudecode.sh Works
    - Session Management
  - Spawning Sub-Agents: The Fast-Track Protocol
    - When to Spawn a Sub-Agent
    - Sub-Agent Spawn Patterns
    - The cleanup: delete Rule
  - Model Policy: Which Agent for Which Task
    - The Hard Rule
    - Why Gemini Sub-Agents Can’t Fix System Issues
  - Sub-Agent Failure Modes
    - Failure Mode 1: The 200K Context Limit
    - Failure Mode 2: Wrong Service Level (User vs System Systemd)

        - Failure Mode 3: Silent Failure on Missing Directory
    - Provider Fallback Chain
        - The Three-Provider Setup
    - Cross-Server Agent Communication
        - The VPN-Based Mesh
    - Agent Mesh Healing: Flock-Based Distributed Coordination
        - The Problem: Two Agents, Same Task
        - Self-Healing: Detecting and Recovering from Stalled Agents
    - Sub-Agent Best Practices: The Runbook
        - Do’s
        - Don’ts
        - Task Description Template
    - Lessons Learned
- PART 5: THE FUTURE
- Chapter 15: Scaling Agent Systems
    - Introduction
    - Part 1: From One Agent to an Agent Mesh
        - The Single-Agent Phase (Days 1–30)
        - Designing the Agent Mesh (Days 30–90)
        - Mesh Governance: Who Owns What
    - Part 2: Horizontal Scaling
        - When to Add a Server
        - The Three-Server Baseline
        - Load Balancing Agent Requests
    - Part 3: Database High-Availability — The Hard Lessons
        - Patroni + etcd: The Architecture
        - The 22-Failover Weekend: A Postmortem
        - Patroni Configuration (Production-Hardened)
        - Backup Strategy: Three Repositories
    - Part 4: Memory and Knowledge Scaling
        - The Problem with Shared Agent Memory
        - Cognee Across Multiple Servers
        - Memory Tiers
    - Part 5: Cost Scaling
        - The Exponential Problem
        - Model Right-sizing
        - The Cron Job Decision
    - Part 6: Memory Management at the OS Level
        - The Swap Problem
    - Summary

- Chapter 16: Lessons Learned — What I Wish I'd Known
    - Introduction
    - The Top 10 Mistakes
        - Mistake #1: Not Setting Budget Alerts (The $500 Lesson)
        - Mistake #2: Trusting Agents with Infrastructure Config (The API Proxy Incident)
        - Mistake #3: Auto-Healing That Costs More Than the Problems It Fixes
        - Mistake #4: Not Reporting Progress (Agent Silence = User Anxiety)
        - Mistake #5: Hardcoding Credentials
        - Mistake #6: Ignoring the Firewall (Server-3 UFW Never Enabled)
        - Mistake #7: Mixing Agent Data (The Cognee Multi-Tenancy Problem)
        - Mistake #8: Over-Engineering Monitoring (The v1→v2→v2.2→v2.3 Iteration Trap)
        - Mistake #9: Using the Wrong Model for the Wrong Task (Gemini for Infrastructure = Fail)
        - Mistake #10: Not Testing Sub-Agent Outputs (Silent Failures)
    - The Human Element
        - AI Generates Fast; Everything Else Takes 10x Longer
        - The Trust Calibration Problem
    - Predictions for the Field
        - MCP Standardization Will Become Table Stakes
        - 2M+ Context Windows Will Change Memory Architecture
        - Adaptive Reasoning Will Replace Static Prompting
    - Final Advice for DevOps Engineers Entering the AI Agent Space
- APPENDICES
- Appendix A: Tools and Frameworks Reference
    - Agent Frameworks
        - CrewAI
        - LangChain
        - AutoGen (Microsoft)
        - Claude Code
        - Gateway Agent (Orchestrator)
    - Infrastructure
        - Docker
        - Traefik
        - HAProxy
        - Patroni
        - etcd
    - Monitoring
        - Prometheus
        - Grafana

        - Alertmanager
        - Loki
        - Tempo
    - Knowledge and Memory
        - Cognee
        - Firecrawl
        - Crawl4AI
        - Neo4j
        - LanceDB
    - Security
        - HashiCorp Vault
        - UFW (Uncomplicated Firewall)
        - WireGuard VPN
    - AI APIs
        - Anthropic (Claude)
        - OpenAI (GPT)
        - Google (Gemini)
        - DeepSeek
        - Mistral
    - Proxy and Load Balancing
        - API Proxy Service
        - API Provider Manager
    - Communication
        - Telegram Bots
        - tmux-based Inter-Agent Messaging
- Appendix B: AI Model Pricing Reference (March 2026)
    - Quick Reference: Price per MTok
    - Claude (Anthropic)
        - Pricing Table
        - Discounts and Multipliers
        - Claude Model Selection Guide
    - OpenAI (GPT)
        - Pricing Table
        - Discounts and Multipliers
        - OpenAI Batch API
    - Google (Gemini)
        - Pricing Table
        - Discounts and Multipliers
        - Free Tier (Google AI Studio)
    - Open Source / Self-Hosted
        - API Pricing (via provider APIs)

    - Self-Hosting Costs
  - Monthly Cost Estimates by Agent Load
    - 1M Tokens/Day (~400 requests/day)
    - 10M Tokens/Day (~4,000 requests/day)
    - 100M Tokens/Day (~40,000 requests/day)
  - Cost Optimization Techniques
    - 1. Prompt Caching
    - 2. Output Token Control
    - 3. Context Window Management
    - 4. Structured Output Schemas
  - Pricing Watch: What Changes and What to Monitor
- Appendix C: Production Checklists
  - 1. Pre-Deployment Checklist
    - Code and Configuration
    - Testing
    - Infrastructure
    - Observability
    - Communication
    - Rollback Plan
  - 2. Security Hardening Checklist
    - Network
    - Authentication and Secrets
    - System
    - TLS and Certificates
    - Agent-Specific
  - 3. Cost Management Checklist
    - Budget Controls
    - Model Right-Sizing
    - Token Efficiency
    - Scheduled Jobs Audit
    - Cost Anomaly Detection
  - 4. Monitoring Setup Checklist
    - Prometheus
    - Alertmanager
    - Grafana
    - Logging (Loki)
  - 5. Incident Response Checklist
    - First 5 Minutes: Triage
    - Diagnosis Commands
    - Containment
    - Resolution

    - Post-Incident (within 48 hours)
- 6. Agent Debugging Checklist
    - Confirm the Problem
    - Inspect the Agent
    - Common Failure Patterns
    - Prompt Debugging
    - Tool Use Debugging
- 7. Model Selection Decision Tree
- Checklist Maintenance

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 with Claude, GPT, Gemini, and open-source models.

Table of Contents

Part 1: Foundations

- Chapter 1: Why This Book Exists
- Chapter 2: What Are AI Agents, Really?
- Chapter 3: Choosing Your Model

Part 2: Architecture

- Chapter 4: Agent Architecture Patterns
- Chapter 5: Context Engineering
- Chapter 6: Tool Use and Function Calling

Part 3: Operations

- Chapter 7: Deployment and Infrastructure
- Chapter 8: Cost Management
- Chapter 9: Monitoring and Observability
- Chapter 10: Debugging AI Agents

Part 4: Advanced Topics

- Chapter 11: Security Hardening
- Chapter 12: AI Image Generation at Scale
- Chapter 13: Knowledge Management

- Chapter 14: Multi-Agent Orchestration

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.

Nearly $500 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 nearly 300 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 over $500. 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 over a dozen

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 multiple geographic regions, backed by cloud providers 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:

- **The gateway agent**: A sister AI agent that monitors my Claude Code sessions, handles browser automation, sends Telegram messages, and can provision resources when I’m asleep
- **The API proxy service**: A local routing gateway that manages over a dozen provider 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 hundreds of 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: Over a Dozen 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 — our API proxy. This proxy sits between our agents and the model providers, handling authentication, routing, rate limiting, and failover. To extend capacity, I had registered 15 accounts with an API provider (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: nearly 300 blog posts, each requiring title generation, outline creation, content generation, and image metadata generation. Four API calls per post. Over 1,000 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 provider 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:

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

I spent six hours on Thursday afternoon doing triage. Account appeals to the provider (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:

```
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 provider 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.

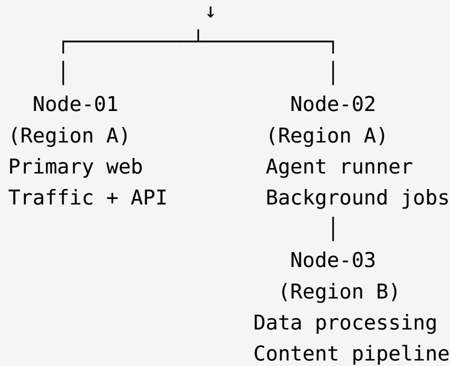
If you are debugging an existing agent system: Skip to Chapter 14 (Debugging War Stories). Then read whatever background chapters address your specific issue.

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 our actual infrastructure. Here is the quick map so you can orient yourself when I mention specific components:



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

Internet → Load Balancer (cloud provider)
                ↓
    ┌───────────┴──────────┐
    │                      │
  Node-01              Node-02
(Region A)           (Region A)
Primary web          Agent runner
Traffic + API        Background jobs
                           │
                       Node-03
                      (Region B)
                    Data processing
                    Content pipeline

Local workstation (me):
  └── API Proxy Service (port 3000)
        ├── Provider Pool A (10 accounts) → Gemini API
        ├── Provider Pool B (5 accounts)  → Monitoring only
        └── Direct API keys (Claude, OpenAI)

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

When I say “the agent SSH’d into Node-01,” this is what I mean. When I say “the proxy rotated to the next provider 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. Our gateway agent has saved hours every week. Our 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.

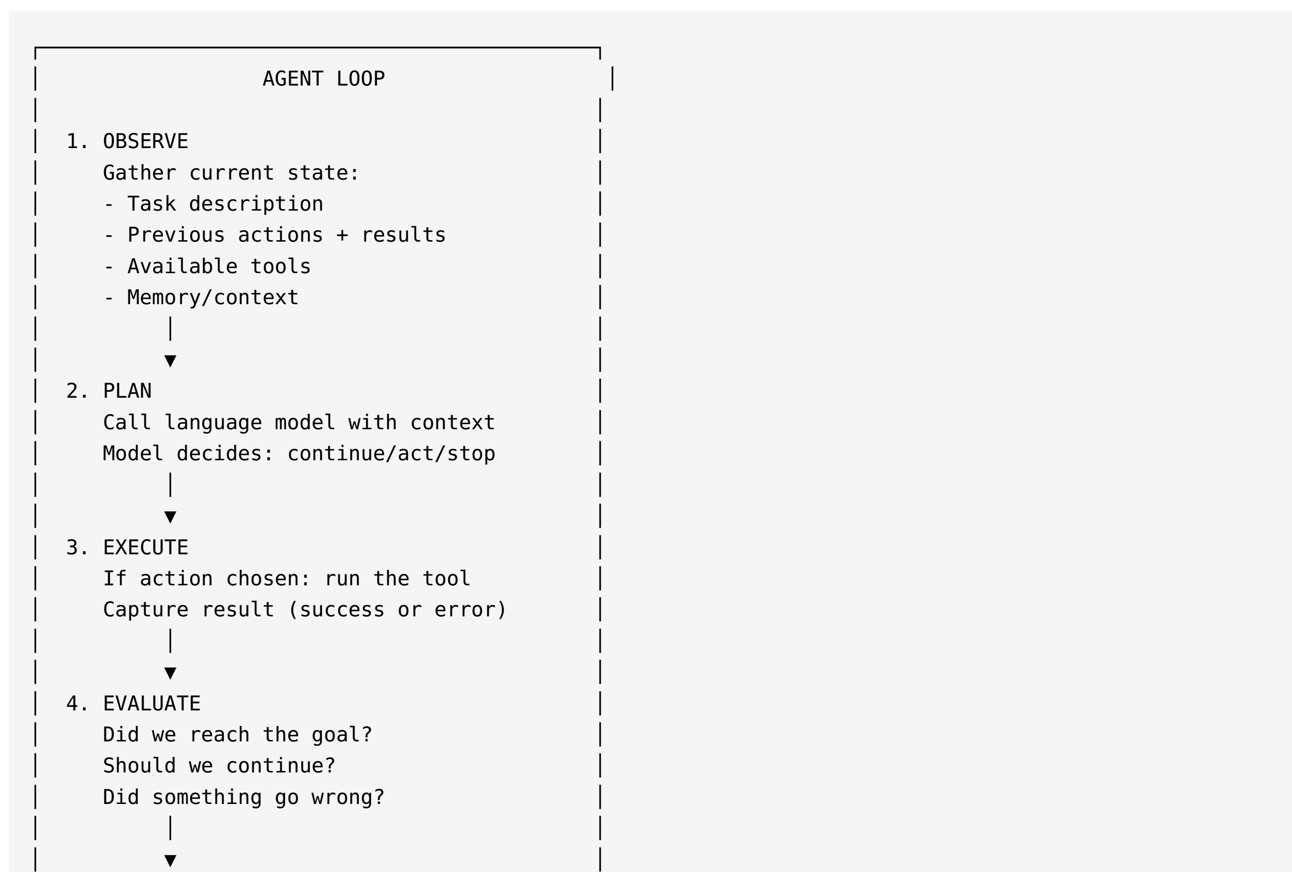
This has a critical implication: **language models do not have state between calls.** Every API call is stateless. The model does not remember your previous call. It only sees what you include in the current request's context window.

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?                 │
│          │                                  │
│          ▼                                  │
```