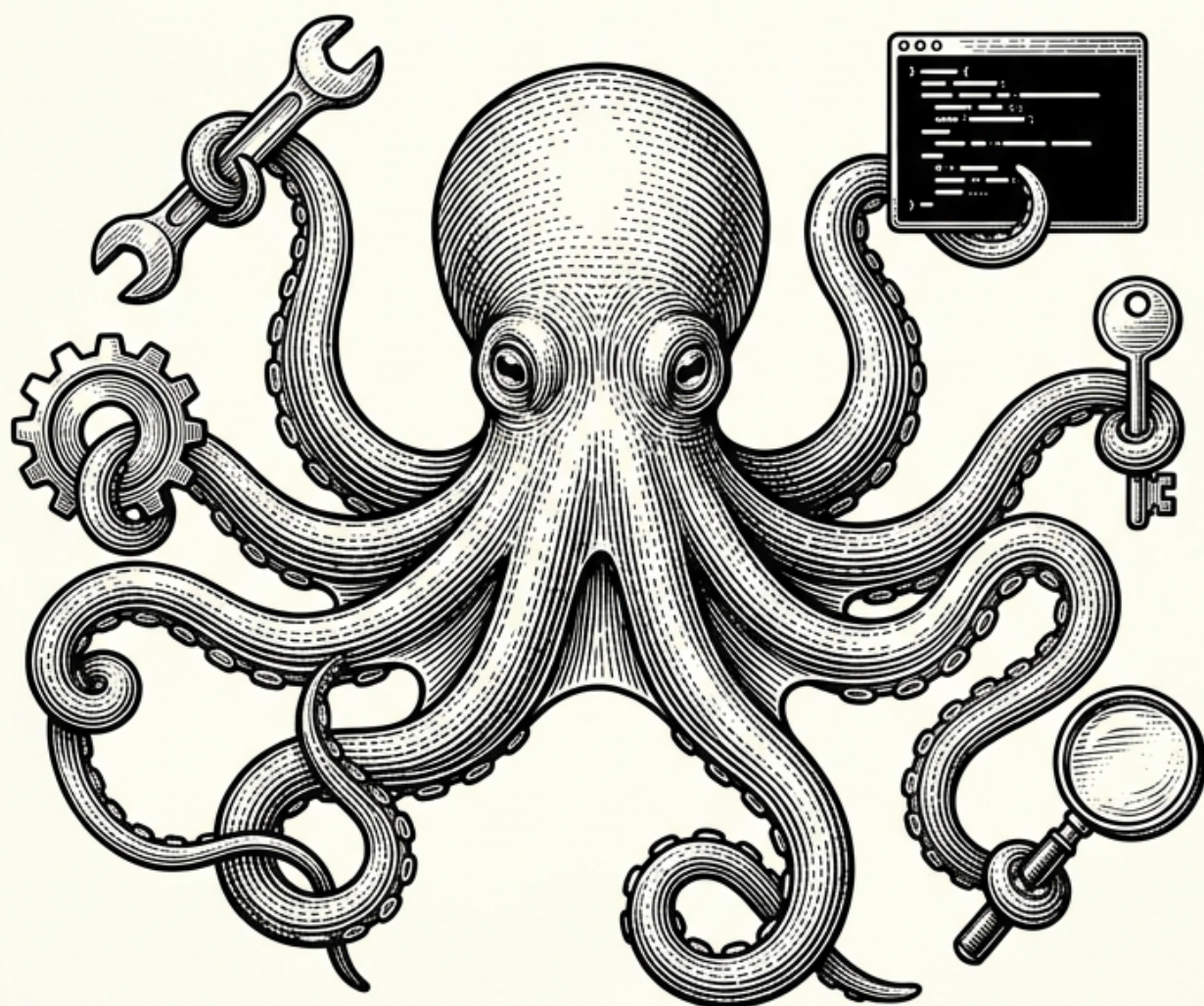


AI Agents in Production

War Stories from a DevOps Engineer



Second Edition
Do Cao Hieu

- AI Agents Trong Môi Trường Production: Câu Chuyện Chiến Trường Từ Một DevOps Engineer
- PHẦN 1: NỀN TẢNG
- Chương 1: Tại Sao Cuốn Sách Này Ra Đời
 - Email Tôi Không Bao Giờ Nghĩ Mình Sẽ Phải Viết
 - Khoảng Cách Giữa Demo Và Thực Tế
 - Người Viết Cuốn Sách Này Là Ai
 - Điều Gì Thay Đổi Từ 2024 Đến 2026
 - Cuộc Cách Mạng Context Window
 - Câu Chuyện Chiến Trường Nền Tảng: Hơn Chục Tài Khoản Bị Suspend Trong Một Ngày
 - Bối Cảnh
 - Điều Đã Xảy Ra
 - Hiệu Ứng Dây Chuyền
 - Thay Đổi Sau Đó
 - Nội Dung Cuốn Sách Này
 - Cuốn Sách Này Không Đề Cập
 - Cuốn Sách Này Dành Cho Ai
 - Một Ghi Chú Về Sự Thành Thật
 - Cách Sử Dụng Cuốn Sách Này
 - Thiết Lập Khiến Mọi Thứ Trở Nên Thực Tế
 - Trước Khi Bắt Đầu
 - Điểm Mấu Chốt
- Chương 2: AI Agent Thực Sự Là Gì?
 - Vấn Đề Định Nghĩa
 - Language Model Thực Sự Là Gì
 - Vòng Lặp Agent
 - Điều Phân Biệt Agent Với Chatbot
 - Các Loại Kiến Trúc Agent
 - Agent Đơn
 - Đa Agent (Orchestrator + Specialist)
 - Mesh / Swarm
 - Kiến Trúc Gateway Agent: Hệ Thống Đa Agent Thực Tế
 - Tool: Đôi Tay Của Agent
 - Nguyên Tắc Thiết Kế Tool
 - Câu Chuyện Chiến Trường: Hệ Thống Agent Mesh Monitor Tự Đốt Mình
 - Thiết Lập
 - Vấn Đề

- Vấn Đề Tệ Hơn
- Quyết Định
- Bài Học
- Agent vs. Chatbot vs. Copilot: Bảng Thực Tế
- Phiên Bản Node.js: Cho Kỹ Sư Server-Side
- Kết Thúc: Điều Quan Trọng Nhất Bạn Chưa Nghĩ Đến
- Điểm Mấu Chốt
- **Chương 3: Chọn Model Của Bạn: Hướng Dẫn Của Kỹ Sư DevOps**
 - Vấn Đề Lựa Chọn Model
 - Bức Tranh Giá Cả (Tháng 3 Năm 2026)
 - Claude 4.x — Anthropic
 - Dòng GPT — OpenAI
 - Gemini — Google
 - Mã Nguồn Mở / Tự Lưu Trữ / Bên Thứ Ba
 - Cách Suy Nghĩ Về Lựa Chọn Model
 - Thang Khả Năng
 - Ma Trận Lựa Chọn
 - Câu Chuyện Chiến Trường: Định Tuyển Task Hạ Tầng Đến Sai Model
 - Thiết Lập
 - Điều Đã Xảy Ra
 - Bài Học
 - Câu Chuyện Chiến Trường: API Proxy Round-Robin Với 15 Tài Khoản Nhà Cung Cấp
 - Kiến Trúc
 - Ngày Tất Cả 15 Tài Khoản Chạm 100% Đồng Thời
 - Câu Chuyện Chiến Trường: Bug Không Khớp Model (HTTP 400 INVALID_ARGUMENT)
 - Thiết Lập
 - Điều Đã Xảy Ra
 - Bài Học Lớn Hơn
 - Xây Dựng Chính Sách Lựa Chọn Model
 - Kiểm Tra Thực Tế Mã Nguồn Mở
 - Theo Dõi Lựa Chọn Model Theo Thời Gian
 - Tham Khảo Nhanh: Quy Tắc Lựa Chọn
 - Điểm Mấu Chốt
- **PHẦN 2: KIẾN TRÚC**
- **Chương 4: Các Mẫu Kiến Trúc Agent**
 - Giới Thiệu
 - Mẫu 1: Agent Đơn (Claude Code Standalone)
 - Khi Nào Nó Hoạt Động

- Ràng Buộc Thực Sự: Context Window
- Anti-Pattern Agent Đơn
- Mẫu 2: Điều Phối Đa Agent
 - Cách Nó Hoạt Động Trong Thực Tế
 - Ngân Sách Context Sub-Agent
 - Mẫu Điều Phối Tôi Dừng
 - Câu Chuyện Chiến Trường: Tắc Nghẽn 15 Agent
 - Vấn Đề Dọn Dẹp
- Mẫu 3: Agent Mesh
 - Tiến Hóa: v1 → v2.3
 - Script Mesh Monitor Đầy Đủ
 - Deployment: systemd Timer Thay Vì Cron
 - Những Gì Mesh Không Thể Sửa
- Mẫu 4: Agent Hướng Sự Kiện
 - Kiến Trúc Cảnh Báo Hai Lớp
 - Tại Sao Hai Lớp?
- Mẫu 5: Kiến Trúc Hybrid
 - Lựa Chọn Model Theo Task
- Hướng Dẫn Quyết Định Kiến Trúc
 - Thuế Phối Hợp
 - Danh Mục Dạng Lỗi
 - Observability Trên Các Kiến Trúc
- Điểm Mấu Chốt
- **Chương 5: Context Engineering — Nghệ Thuật Tối Ưu Token**
 - Giới Thiệu
 - Context Engineering Là Gì?
 - Cấu Trúc Tiêu Thụ Token
 - Khủng Hoảng: Tài Khoản Provider Bị Tạm Đình Chỉ
 - Kiểm Toán File Context
 - Kiến Trúc Context Ba Tầng
 - Tầng 1: Luôn Inject (System Prompt)
 - Tầng 2: Tham Chiếu Theo Yêu Cầu (Không Inject)
 - Tầng 3: Đã Lưu Trữ / Đã Xóa
 - CLAUDE.md và AGENTS.md Như Công Cụ Context Engineering
 - CLAUDE.md Nên Chứa Gì
 - Tỷ Lệ Instruction-to-Pointer
 - Tối Ưu Heartbeat: Từ 3 Phút → 30 Phút
 - Heartbeat Thực Sự Cần Gì
 - Vấn Đề Orphan Session
 - Cách Tìm Orphaned Session

- Phòng Ngừa
 - Context Xuyên Phiên: Tại Sao Absolute Path Là Bắt Buộc
 - Thất Bại Super-Prompt: Tại Sao Task Lớn Cần Chia Thành Giai Đoạn
 - Giải Pháp Chia Giai Đoạn
 - Vệ Sinh Thư Mục Skills: Dọn Dẹp 145 → 105
 - Máy Tính Ngân Sách Context
 - Tiện Ích Đếm Token
 - Checklist Tối Ưu Thực Tế
 - Sự Thay Đổi Tư Duy
 - Điểm Máu Chốt
- **Chương 6: Tool Use và Function Calling**
 - Giới Thiệu
 - Cách Tool Use Hoạt Động: Vòng Lặp Request-Response
 - Claude `tool_use` vs OpenAI / Gemini `function_calling`
 - Claude (Anthropic)
 - OpenAI
 - Gemini (Google)
 - Tóm Tắt So Sánh Nền Tảng
 - Thiết Kế Tool Schema Tốt
 - Bốn Thuộc Tính Của Tool Schema Tốt
 - Triển Khai Tool Cấp Production
 - Tool Use Với Interleaved Thinking (Claude Opus 4.6)
 - MCP: Model Context Protocol
 - Cách MCP Hoạt Động
 - Thiết Lập MCP Trong Claude Code
 - Ví Dụ Thực Tế: Tích Hợp Cognition MCP
 - Triển Khai Cognition MCP Server
 - Viết MCP Server Của Riêng Bạn
 - Thực Thi Tool Song Song
 - Câu Chuyện Chiến Trường: Agent Tự Chặt Chân Mình
 - Bối Cảnh
 - Điều Đã Xảy Ra
 - Bài Học: Guardrail Phải Rõ Ràng
 - Quản Lý API Key: Lỗi Đánh Máy Proxy Key Tốn Hàng Giờ
 - Pattern Tool Use Cho Production
 - Pattern: Read-Before-Write
 - Pattern: Dry-Run Trước
 - Pattern: Tool Idempotent
 - Điểm Máu Chốt

- PHÂN 3: VẬN HÀNH
- Chương 7: Triển Khai AI Agent Lên Production
 - Topology Server
 - Kiểm Tra Thực Tế Process Manager: Docker vs Systemd vs PM2
 - Kiểm Kế Dịch Vụ (Kiểm Tra Thực Tế Server-1)
 - Khi Nào Nên Dừng Docker
 - Khi Nào Nên Dừng Systemd
 - Khi Nào Nên Dừng PM2 (Và Khi Nào Không)
 - Traefik Với docker-socket-proxy (Cách Đúng)
 - VPN Binding Cho Các Port Nhạy Cảm
 - HAProxy Cho Database Connection Pooling
 - Câu Chuyện Chiến Trường #1: Bug Cô Lập Mạng Firecrawl
 - Điều Đã Xảy Ra
 - Nguyên Nhân Gốc
 - Cách Sửa
 - Bài Học
 - Câu Chuyện Chiến Trường #2: Server Không Có Firewall
 - Điều Đã Xảy Ra
 - Đánh Giá Thiệt Hại
 - Quy Trình Hardening An Toàn
 - Những Gì Chúng Tôi Thêm Vào Checklist Kiểm Toán
 - Pattern Docker Compose Cho Agent Service
 - Pattern 1: Agent Với Mạng Cô Lập
 - Pattern 2: Multi-Container Agent Stack
 - Pattern 3: Systemd Unit Cho Agent Process
 - Quản Lý Bí Mật
 - Health Check Và Quản Lý Phụ Thuộc
 - Checklist Triển Khai
 - Tóm Tắt
- Chương 8: Quản Lý Chi Phí — Đừng Phá Sản Khi Chạy AI Agent
 - Sự Cố Gemini \$500
 - Điều Đã Xảy Ra
 - Điều Lẽ Ra Phải Xảy Ra
 - Cách Lấy Lại Tiền
 - Cảnh Báo Ngân Sách Không Phải Tùy Chọn
 - Cảnh Báo Ngân Sách Google Cloud
 - Ngân Sách Anthropic Claude
 - Tín Dụng Dừng Thử Và Các Model Không Áp Dụng

- Khủng Hoảng Tối Ưu Token
 - Vấn Đề: Lượng Token Cơ Sở
 - Tối Ưu Hóa
 - Tối Ưu Khoảng Cách Heartbeat
 - Xóa Tất Cả Cron Job AI Tự Động
 - Orphan tmux Session: Kẻ Giết Quota Thâm Lặng
 - API Proxy Cục Bộ: Quota Miễn Phí Qua Account Rotation
 - Cách Hoạt Động
 - Thực Tế Quota
 - Đánh Đổi Độ Trễ
 - Chiến Lược Phân Cấp Model
 - Batch API: Giảm 50% Chi Phí
 - Dashboard Giám Sát Chi Phí
 - Chi Phí Của “Chỉ Kiểm Tra”
 - Script Giám Sát Ngân Sách
 - Các Quy Tắc Quản Lý Chi Phí Chính
- **Chương 9: Giám Sát AI Agent Trong Môi Trường Production**
 - Những Gì Bạn Thực Sự Cần Giám Sát
 - Monitoring Stack
 - Cấu Hình Prometheus
 - Alert Rules Cho Agent Services
 - Cấu Hình Alertmanager
 - Docker Events Listener: Cảnh Báo Container Thời Gian Thực
 - Agent Mesh Monitor: Kiểm Tra Sức Khỏe Đa Server
 - Script Kiểm Tra Sức Khỏe Theo Từng Server
 - Câu Chuyện Thực Tế #1: Kiểm Tra Sai Service
 - Chuyện Gì Đã Xảy Ra
 - Cách Sửa
 - Bài Học Rút Ra
 - Câu Chuyện Thực Tế #2: Vòng Lặp Crash 102 Lần Restart
 - Chuyện Gì Đã Xảy Ra
 - Nguyên Nhân Gốc Rễ: Hai Process Đang Chạy
 - Điều Đó Xảy Ra Như Thế Nào
 - Cách Sửa
 - Bài Học Rút Ra
 - Neural Memory Audit: Khi Agent Có Sức Khỏe Riêng
 - Các Memory Health Metric Cần Theo Dõi
 - Cấu Trúc Grafana Dashboard
 - Dashboard 1: Infrastructure Overview (Tổng Quan Hạ Tầng)
 - Dashboard 2: AI Agent Metrics

- Dashboard 3: Logs (Loki)
- Dashboard 4: Alerts History (Lịch Sử Alert)
- Tổng Hợp Log Với Loki
- Checklist Monitoring Hoàn Chỉnh
- Tóm Tắt
- **Chương 10: Debug AI Agent — Khi Mọi Thứ Đi Sai**
 - Thay Đổi Tư Duy Khi Debug
 - Các giả định debug truyền thống không áp dụng được cho agent:
 - Agent Debugging Stack
 - Failure Mode 1: Agent Sửa Đổi Chính Cơ Sở Hạ Tầng Của Nó
 - Sự Cố
 - Tại Sao Điều Này Xảy Ra
 - Phòng Ngừa
 - Khôi Phục
 - Failure Mode 2: Lỗi Đánh Máy API Key và Lỗi Credential
 - Sự Cố
 - Debug Vấn Đề Xác Thực
 - Pattern: Credential Không Nhất Quán Giữa Các File
 - Failure Mode 3: Lỗi Permission
 - Các Lỗi Permission Phổ Biến Trong Bối Cảnh Agent
 - Xây Dựng Permission Diagnostic Tool
 - Failure Mode 4: Rate Limit
 - Vấn Đề Vertex AI 429
 - Pattern Xử Lý Rate Limit
 - API Proxy Tự Động Chuyển Khi Hết Quota
 - Failure Mode 5: Agent Không Báo Cáo Tiến Trình
 - Vấn Đề
 - Pattern Ngăn Ngừa Điều Này
 - Failure Mode 6: Vượt Quá Context Limit Giữa Task
 - Vấn Đề
 - Phòng Ngừa: Quản Lý Context Budget
 - Pattern Chunking
 - Failure Mode 7: Model Mismatch
 - Sự Cố
 - Phát Hiện Model Mismatch
 - Câu Chuyện Thực Tế: SSR CSS Hash Mismatch
 - Chuyện Gì Đã Xảy Ra
 - Cách Sửa
 - Bài Học Rút Ra

- Câu Chuyện Thực Tế: Config Sync Trên Ba Server
 - Chuyện Gì Đã Xảy Ra
 - Chẩn Đoán
 - Phòng Ngừa: Config Sync Script
 - Giải Pháp Dài Hạn Tốt Hơn: Config Tập Trung
- Câu Chuyện Thực Tế: EtcD Split-Brain
 - Chuyện Gì Đã Xảy Ra
 - Cách Sửa
- Câu Chuyện Thực Tế: Lỗi Reasoning Parameter
 - Chuyện Gì Đã Xảy Ra
 - Cách Sửa
- Câu Chuyện Thực Tế: Claude Code Kẹt Ở Trust Dialog
 - Chuyện Gì Đã Xảy Ra
 - Cách Sửa (Ngắn Hạn)
 - Cách Sửa (Dài Hạn)
- Câu Chuyện Thực Tế: Sub-Agent Silent Failure
 - Chuyện Gì Đã Xảy Ra
 - Điều Thực Sự Đã Xảy Ra
 - Phòng Ngừa: Bước Xác Minh Trong Sub-Agent Task
- Debugging Checklist
- Tóm Tắt

• PHẦN 4: CHỦ ĐỀ NÂNG CAO

• Chương 11: Tăng Cường Bảo Mật Cho Cơ Sở Hạ Tầng AI Agent

- Ngày AI Trở Thành Lỗ Hổng Bảo Mật Lớn Nhất Của Chúng Tôi
- Threat Model Đã Thay Đổi
- Phần 1: Quản Lý Vault Token
 - Vấn Đề Root Token
 - Tự Động Hóa Xoay Token
 - Quy Trình Vault Seal/Unseal
- Phần 2: Tăng Cường Network — Các Port Không Bao Giờ Nên Công Khai
 - Phát Hiện
 - Bật UFW Từ Đầu (Không Khóa Bản Thân Ra Ngoài)
 - Bind Service Nhạy Cảm Vào VPN Interface
 - Cloud ARM Server: Dọn Dẹp NFS/rpcbind
- Phần 3: Tăng Cường SSH
 - PermitRootLogin và PasswordAuthentication
 - Xoay SSH Key Khi Onboarding Máy Mới
 - Restart Service Sau Khi Xoay Password
- Phần 4: Credential Plaintext Trong Config File
 - Patroni.yml — Mật Khẩu Phơi Bày

- Credential Hardcode Trong Application Code
 - Phần 5: Docker Security — Vấn Đề docker.sock
 - Traefik và docker.sock
 - Phần 6: HAProxy Stats — Mật Khẩu Mặc Định Không Ai Thay Đổi
 - Phần 7: Vault Policy Cho Agent Access
 - Phần 8: AI Agent Security Checklist
 - Trước Khi Deploy
 - Baseline Hạ Tầng
 - Vận Hành
 - Bài Học Rút Ra
- **Chương 12: AI Image Generation ở Quy Mô Lớn**
 - Vấn Đề Xử Lý Hàng Loạt Ở Quy Mô Lớn
 - Kiến Trúc: Xử Lý Hàng Loạt Ở Quy Mô Lớn
 - Điểm Xuất Phát: Những Gì Chúng Tôi Có
 - Chạm Quota Giữa Batch
 - Lỗi
 - Lựa Chọn 1: Yêu Cầu Tăng Quota (Mất Vài Ngày)
 - Lựa Chọn 2: Hot-Swap API Key Trên Nhiều Project
 - Thảm Họa Delimiter
 - Lỗi Xuất Hiện Như Thế Nào
 - Cách Sửa: Tự Động Phát Hiện Và Chuyển Sang TAB Delimiter
 - Checkpoint và Resume: Không Bao Giờ Bắt Đầu Lại Từ Đầu
 - Nguyên Tắc
 - Resume Sau Khi Thất Bại
 - Thí Nghiệm Logo: Khi AI Không Phải Công Cụ Phù Hợp
 - Đề Xuất
 - Rate Limit: Nút Cổ Chai textembedding-gecko
 - Tình Huống
 - Yêu Cầu Tăng Quota
 - Veo 3: Video Generation
 - Thí Nghiệm
 - Điều Thực Sự Đã Xảy Ra
 - POD Skill: Năm Phiên Bản Script
 - Bài Học Rút Ra
 - **Chương 13: Quản Lý Tri Thức: RAG, Đồ Thị, và Bộ Nhớ**
 - Tại Sao RAG Đơn Thuần Là Chưa Đủ
 - Tổng Quan Kiến Trúc
 - Bộ Công Nghệ
 - Tại Sao Server-3?

- Cài Đặt Cognition
 - Cài Đặt và Cấu Hình
 - Docker Compose cho Neo4j
 - Tổ Chức Dataset
 - Bốn Dataset
 - Tại Sao Cô Lập Dataset Quan Trọng
 - Nhập 400 Cuốn Sách DevOps
 - Pipeline Nhập Liệu
 - Câu Chuyện Thực Tế: Con Bão 409 Conflict
 - Câu Chuyện Thực Tế: Giới Hạn Tốc Độ Vertex AI Làm Đình Trệ Mọi Thứ
 - MCP Server: Tri Thức Được Truy Cập Bởi Agent
 - Cấu Hình MCP
 - Cấu Hình Claude Code MCP
 - Crawl4AI vs Firecrawl: Chọn Công Cụ Thu Thập Web
 - Firecrawl
 - Crawl4AI
 - Ma Trận Quyết Định
 - Bộ Nhớ Thần Kinh: Đồ Thị Trông Như Thế Nào
 - Bài Học Rút Ra
- **Chương 14: Điều Phối Đa Agent Trong Thực Tế**
 - Vấn Đề Với Một Agent
 - Kiến Trúc: Mẫu Gateway Agent
 - Thiết Kế Cốt Lõi
 - Giao Tiếp Qua Shared Workspace
 - Giao Tiếp Dựa Trên Tmux: Gửi Tổ Hợp Phím Giữa Các Phiên
 - Cách ping-claudecode.sh Hoạt Động
 - Quản Lý Phiên
 - Tạo Sub-Agent: Giao Thức Fast-Track
 - Khi Nào Tạo Sub-Agent
 - Các Mẫu Tạo Sub-Agent
 - Quy Tắc cleanup: delete
 - Chính Sách Model: Agent Nào Cho Tác Vụ Nào
 - Quy Tắc Cứng
 - Tại Sao Gemini Sub-Agent Không Thể Sửa Vấn Đề Hệ Thống
 - Các Chế Độ Thất Bại Của Sub-Agent
 - Chế Độ Thất Bại 1: Giới Hạn Context 200K
 - Chế Độ Thất Bại 2: Cấp Độ Service Sai (User vs System Systemd)
 - Chế Độ Thất Bại 3: Thất Bại Thâm Lặng Khi Thiếu Thư Mục
 - Chuỗi Fallback Provider
 - Thiết Lập Ba Provider

- Giao Tiếp Agent Đa Server
 - Mesh Dựa Trên VPN
- Phục Hồi Mesh Agent: Điều Phối Phân Tán Dựa Trên Flock
 - Vấn Đề: Hai Agent, Cùng Tác Vụ
 - Tự Phục Hồi: Phát Hiện và Khôi Phục Từ Các Agent Bị Treo
- Thực Hành Tốt Nhất Cho Sub-Agent: Runbook Vận Hành
 - Nên Làm
 - Không Nên Làm
 - Template Mô Tả Tác Vụ
- Bài Học Rút Ra
- **PHẦN 5: TƯƠNG LAI**
- **Chương 15: Mở Rộng Quy Mô Hệ Thống Agent**
 - Giới Thiệu
 - Phần 1: Từ Một Agent Đến Một Mesh Agent
 - Giai Đoạn Một Agent (Ngày 1-30)
 - Thiết Kế Mesh Agent (Ngày 30-90)
 - Quản Trị Mesh: Ai Sở Hữu Cái Gì
 - Phần 2: Mở Rộng Ngang
 - Khi Nào Thêm Server
 - Baseline Ba Server
 - Cân Bằng Tải Các Yêu Cầu Agent
 - Phần 3: Tính Khả Dụng Cao Của Database — Các Bài Học Khó
 - Patroni + etcd: Kiến Trúc
 - Cuối Tuần 22 Failover: Một Postmortem
 - Cấu Hình Patroni (Hardened cho Production)
 - Chiến Lược Backup: Ba Kho Lưu Trữ
 - Phần 4: Mở Rộng Bộ Nhớ và Tri Thức
 - Vấn Đề Với Bộ Nhớ Agent Chung
 - Cognition Trên Nhiều Server
 - Các Tầng Bộ Nhớ
 - Phần 5: Mở Rộng Chi Phí
 - Vấn Đề Lũy Thừa
 - Định Cỡ Model Đúng
 - Quyết Định Về Cron Job
 - Phần 6: Quản Lý Bộ Nhớ Ở Cấp Độ OS
 - Vấn Đề Swap
 - Tóm Tắt
- **Chương 16: Bài Học Rút Ra — Những Điều Tôi Ước Mình Biết Trước**
 - Giới Thiệu

- Top 10 Sai Lầm
 - Sai Lầm #1: Không Đặt Cảnh Báo Ngân Sách (Bài Học \$500)
 - Sai Lầm #2: Tin Tưởng Agent Với Cấu Hình Infrastructure (Sự Cố API Proxy)
 - Sai Lầm #3: Tự Phục Hồi Tốn Hơn Là Các Vấn Đề Nó Sửa
 - Sai Lầm #4: Không Báo Cáo Tiến Trình (Im Lặng Của Agent = Lo Lắng Người Dùng)
 - Sai Lầm #5: Hardcode Credentials
 - Sai Lầm #6: Bỏ Qua Firewall (Server-3 UFW Chưa Bao Giờ Được Bật)
 - Sai Lầm #7: Trộn Dữ Liệu Agent (Vấn Đề Multi-Tenancy Cognee)
 - Sai Lầm #8: Over-Engineering Monitoring (Bẫy Lặp v1→v2→v2.2→v2.3)
 - Sai Lầm #9: Dùng Model Sai Cho Tác Vụ Sai (Gemini Cho Infrastructure = Thất Bại)
 - Sai Lầm #10: Không Kiểm Tra Output Sub-Agent (Thất Bại Thâm Lặng)
- Yếu Tố Con Người
 - AI Tạo Nhanh; Mọi Thứ Khác Mất Lâu Gấp 10 Lần
 - Vấn Đề Hiệu Chính Tin Tưởng
- Dự Đoán Cho Lĩnh Vực
 - Tiêu Chuẩn Hóa MCP Sẽ Trở Thành Table Stakes
 - Context Window 2M+ Sẽ Thay Đổi Kiến Trúc Bộ Nhớ
 - Suy Luận Thích Nghi Sẽ Thay Thế Prompting Tĩnh
- Lời Khuyên Cuối Cho DevOps Engineers Bước Vào Không Gian AI Agent

• PHỤ LỤC

• Phụ Lục A: Tài Liệu Tham Khảo Công Cụ và Framework

- Framework Agent
 - CrewAI
 - LangChain
 - AutoGen (Microsoft)
 - Claude Code
 - Gateway Agent (Orchestrator)
- Infrastructure
 - Docker
 - Traefik
 - HAProxy
 - Patroni
 - etcd
- Monitoring
 - Prometheus
 - Grafana
 - Alertmanager
 - Loki
 - Tempo

- Tri Thức và Bộ Nhớ
 - Cognition
 - Firecrawl
 - Crawl4AI
 - Neo4j
 - LanceDB
- Bảo Mật
 - HashiCorp Vault
 - UFW (Uncomplicated Firewall)
 - WireGuard VPN
- AI API
 - Anthropic (Claude)
 - OpenAI (GPT)
 - Google (Gemini)
 - DeepSeek
 - Mistral
- Proxy và Cân Bằng Tải
 - API Proxy Service
 - API Provider Manager
- Giao Tiếp
 - Telegram Bot
 - Nhắn Tin Liên Agent Dựa Trên tmux
- Phụ Lục B: Tài Liệu Tham Khảo Giá AI Model (Tháng 3 Năm 2026)
 - Tham Khảo Nhanh: Giá Mỗi MTok
 - Claude (Anthropic)
 - Bảng Giá
 - Giảm Giá và Hệ Số Nhân
 - Hướng Dẫn Lựa Chọn Model Claude
 - OpenAI (GPT)
 - Bảng Giá
 - Giảm Giá và Hệ Số Nhân
 - Batch API OpenAI
 - Google (Gemini)
 - Bảng Giá
 - Giảm Giá và Hệ Số Nhân
 - Free Tier (Google AI Studio)
 - Open Source / Tự Host
 - API Pricing (qua API provider)
 - Chi Phí Tự Host
 - Ước Tính Chi Phí Hàng Tháng Theo Tải Agent
 - 1M Token/Ngày (~400 yêu cầu/ngày)

- 10M Token/Ngày (~4.000 yêu cầu/ngày)
- 100M Token/Ngày (~40.000 yêu cầu/ngày)
- Kỹ Thuật Tối Ưu Chi Phí
 - 1. Prompt Caching
 - 2. Kiểm Soát Output Token
 - 3. Quản Lý Context Window
 - 4. Schema Output Có Cấu Trúc
- Theo Dõi Giá: Những Gì Thay Đổi và Những Gì Cần Giám Sát

- Phụ lục C: Checklist Môi Trường Production

- 1. Checklist Trước Khi Deploy
 - Code và Configuration
 - Testing
 - Infrastructure
 - Observability
 - Thông Báo
 - Kế Hoạch Rollback
- 2. Checklist Bảo Mật Hệ Thống
 - Network
 - Authentication và Secrets
 - System
 - TLS và Certificates
 - Đặc Thù Agent
- 3. Checklist Quản Lý Chi Phí
 - Kiểm Soát Budget
 - Chọn Model Phù Hợp
 - Hiệu Quả Token
 - Kiểm Tra Scheduled Job
 - Phát Hiện Cost Anomaly
- 4. Checklist Thiết Lập Monitoring
 - Prometheus
 - Alertmanager
 - Grafana
 - Logging (Loki)
- 5. Checklist Xử Lý Sự Cố
 - 5 Phút Đầu: Triage
 - Lệnh Chẩn Đoán
 - Kiểm Chế Sự Cố
 - Giải Quyết
 - Sau Sự Cố (trong vòng 48 giờ)
- 6. Checklist Debug Agent
 - Xác Nhận Vấn Đề

- Kiểm Tra Agent
- Các Pattern Lỗi Phổ Biến
- Debug Prompt
- Debug Tool Use
- 7. Cây Quyết Định Chọn Model
- Bảo Trì Checklist

AI Agents Trong Môi Trường Production: Câu Chuyện Chiến Trờng Từ Một DevOps Engineer

Ấn Bản Thứ Hai — Tháng 3, 2026

Hướng dẫn thực hành xây dựng, triển khai và vận hành AI agents trong môi trường production. Dựa trên kinh nghiệm thực tế vận hành hệ thống multi-agent với Claude, GPT, Gemini và các model mã nguồn mở.

PHẦN 1: NỀN TẢNG

Chương 1: Tại Sao Cuốn Sách Này Ra Đời

“Cách tốt nhất để học một điều gì đó là bị nó đốt trước.” — Mọi kỹ sư DevOps đã từng sống sót sau hóa đơn cloud \$500

Email Tôi Không Bao Giờ Nghĩ Mình Sẽ Phải Viết

Lúc 2 giờ 47 sáng một ngày thứ Ba, điện thoại tôi rung lên. Không phải cảnh báo PagerDuty, không phải thông báo server ngã — mà là cảnh báo thanh toán từ Google Cloud. Loại thông báo khiến dạ dày bạn như chìm xuống trước khi bạn kịp đọc con số.

Gần \$500 trong 48 giờ.

Tôi nhìn chăm chăm vào màn hình cả phút. API Gemini. Một tiến trình tôi đã để chạy ba ngày trước — thứ tôi nghĩ là một pipeline tạo ảnh gọn gàng, giới hạn rõ ràng cho gần 300 bài blog — đã thoát ra khỏi vòng kiểm soát và đang vui vẻ lặp đi lặp lại, tái tạo, xử lý lại, đốt token như lò nung không có nhiệt kế.

Sáng hôm sau nó vượt \$500. Phản hồi của Google, sau một ticket hỗ trợ và lời giải thích thành thật đến xấu hổ, là “tín dụng miễn trừ một lần.” Mười một chữ vừa cứu tiền tôi vừa phá nát lòng tự trọng của tôi.

Sự cố đó không phải nguyên nhân tôi viết cuốn sách này. Nhưng nó làm rõ tại sao cuốn sách cần tồn tại.

Khoảng Cách Giữa Demo Và Thực Tế

Năm 2024, mọi nhà cung cấp AI đều chiếu cho bạn xem cùng một demo. Một lập trình viên gõ yêu cầu. Agent suy nghĩ một chút. Code xuất hiện. Các bài test đều pass. Sao băng trên trời. Khán giả vỗ tay.

Điều demo không bao giờ cho bạn thấy: - Điều gì xảy ra khi agent lập 6 tiếng với một đầu vào bị lỗi - Cách debug một hệ thống đa agent khi ba agent không đồng ý với nhau - “100% quota exceeded” trông như thế nào khi hơn chục tài khoản API đồng thời chạm giới hạn - Biểu cảm trên mặt bạn khi một tiến trình không kiểm soát đốt \$500 trong 48 giờ - Tại sao đôi khi model \$0.02/M token lại hợp lý hơn model \$25/M token - Cách thực sự theo dõi, giới hạn tốc độ và kiểm soát chi phí cho các tiến trình tự động trong môi trường production

Cuốn sách này lấp đầy khoảng cách đó. Không phải khoảng cách của demo — mà là khoảng cách của 3 giờ sáng.

Người Viết Cuốn Sách Này Là Ai

Tôi vận hành hạ tầng đa server trải rộng nhiều vùng địa lý, được hỗ trợ bởi các nhà cung cấp cloud và đặt phía sau load balancer. Stack này phục vụ lưu lượng thực, xử lý người dùng thực, và từ giữa năm 2024, chạy AI agent như workload production hạng nhất.

Không phải “agent” theo nghĩa chatbot có bộ nhớ. Mà là hệ thống tự động thực sự:

- **Gateway agent:** Một AI agent chị em theo dõi các phiên Claude Code của tôi, xử lý tự động hóa trình duyệt, gửi tin nhắn Telegram, và có thể cấp phát tài nguyên khi tôi đang ngủ
- **Dịch vụ proxy API:** Một cổng định tuyến cục bộ quản lý hơn chục tài khoản API của nhà cung cấp theo phương thức round-robin, với tự động failover khi bất kỳ tài khoản nào chạm quota
- **Content pipeline:** Các batch agent tạo bài blog, ảnh và metadata trên hàng trăm bài viết
- **Infrastructure agent:** Các hệ thống có thể SSH vào server, kiểm tra log, khởi động lại dịch vụ, và — quan trọng là — cũng có thể mắc sai lầm

Tôi đã vận hành hạ tầng AI agent trong production suốt mười tám tháng. Tôi đã mắc gần như mọi sai lầm mà cuốn sách này mô tả. Một số sai lầm tôi mắc đến hai lần.

Nền tảng của tôi là DevOps và SRE. Tôi tư duy theo uptime SLO, cost-per-request và blast radius. Tôi quan tâm đến những thứ như: điều gì xảy ra khi cái này hỏng lúc 3 giờ sáng? Ai trả cho sai lầm đó? Làm sao tôi nhận được cảnh báo khi nó sai?

Cuốn sách này được viết từ góc nhìn đó.

Điều Gì Thay Đổi Từ 2024 Đến 2026

Khi tôi bắt đầu xây dựng hạ tầng agent năm 2024, bức tranh trông như thế này:

Tính năng	2024	2026
Model tốt nhất	Claude 3.5 Sonnet	Claude Opus 4.6
Context window	128K token	200K tiêu chuẩn, 1M beta
Độ tin cậy của tool use	~70% tỷ lệ thành công	95%+ với task rõ ràng
Framework đa agent	Thử nghiệm	Sẵn sàng cho production
Chi phí (model tốt nhất)	\$15/MTok input	\$5/MTok input (Opus 4.6)
Công cụ debug agent	Không tồn tại	Đang hình thành
Quản lý quota	Thủ công	Có thể tự động hóa một phần

Các model đã cải thiện vượt bậc. Claude 3.5 → 4.6 không chỉ là cú bump marketing — đó là bước nhảy vọt về chất lượng suy luận, độ chính xác của tool use và tính nhất quán trong ngữ cảnh dài. GPT-4 → 5.x mang lại cải tiến tương tự. Gemini từ một model đầy hứa hẹn nhưng không ổn định đã trở thành thứ gì đó thực sự cạnh tranh về giá-chất lượng.

Nhưng đây là điều không thay đổi: **AI agent trong production vẫn hỏng theo đủ mọi cách mà phần mềm production hỏng**, cộng thêm một loại lỗi hoàn toàn mới đặc thù của AI mà năm 2024 chưa ai có tên gọi.

Các công cụ cho observability, kiểm soát chi phí và điều phối đa agent đã cải thiện, nhưng vẫn đang ở giai đoạn “Kubernetes đầu năm 2015”. Bạn có thể làm cho nó hoạt động. Nhưng cần có sự khéo léo.

Cuộc Cách Mạng Context Window

Một thay đổi cụ thể đáng được chú ý vì nó đã định hình lại cách agent hoạt động.

Năm 2024, 128K token cảm giác rất lớn. Nhưng thực tế, khi bạn thêm system prompt, lịch sử hội thoại, định nghĩa tool và vài vòng kết quả tool, bạn đã ở 80K và bắt đầu lo lắng. Áp lực context buộc phải liên tục đánh đổi: tóm tắt nhiều hơn, cắt lịch sử sớm hơn, chia task thành các phần nhỏ hơn.

Năm 2026, context window 1M token thay đổi cả bài toán. Bạn có thể đưa cho agent toàn bộ codebase, lịch sử hội thoại đầy đủ và kết quả tool dày đặc mà không chạm giới hạn. Điều này cho phép:

- **Chạy tự động dài hơn:** Agent có thể làm việc trên task phức tạp, nhiều bước mà không mất context
- **Bộ nhớ phong phú hơn:** Lịch sử hội thoại đầy đủ thay vì bản tóm tắt nén
- **Debug tốt hơn:** Agent có thể xem lại quyết định trước đó của mình trước khi hành động
- **Liên tục giữa các phiên:** Các file context trải dài nhiều ngày làm việc

Đánh đổi là chi phí. Context 1M token tốn tiền mỗi khi bạn gửi yêu cầu. Context engineering — nghệ thuật quyết định cái gì đi vào context window và cái gì được để ra ngoài — đã trở thành một trong những kỹ năng quan trọng nhất trong phát triển agent. Chúng ta sẽ đề cập chi tiết trong Chương 7.

Câu Chuyện Chiến Trường Nền Tảng: Hơn Chục Tài Khoản Bị Suspend Trong Một Ngày

Trước khi đi tiếp, hãy để tôi kể cho bạn nghe về ngày tôi thay đổi hoàn toàn cách nghĩ về hạ tầng AI.

Bối Cảnh

Tôi chạy workload AI thông qua một dịch vụ proxy cục bộ — API proxy của chúng tôi. Proxy này nằm giữa các agent và nhà cung cấp model, xử lý xác thực, định tuyến, giới hạn tốc độ và failover. Để mở rộng năng lực, tôi đã đăng ký 15 tài khoản với một nhà cung cấp API (một bộ tổng hợp truy cập API cung cấp tín dụng Gemini API miễn phí/trợ cấp).

Hệ thống rất thanh lịch: agent gọi vào proxy cục bộ, proxy round-robin qua 15 tài khoản, và khi bất kỳ tài khoản nào chạm giới hạn quota, proxy tự động chuyển sang tài khoản khả dụng tiếp theo. 15 tài khoản nghĩa là giới hạn tốc độ hiệu quả gấp 15 lần. Thông minh đúng không?

Điều Đã Xảy Ra

Vào một sáng thứ Năm tháng Hai năm 2026, tôi khởi động một job tạo nội dung batch lớn: gần 300 bài blog, mỗi bài cần tạo tiêu đề, tạo outline, tạo nội dung và tạo metadata ảnh. Bốn lời gọi API mỗi bài. Hơn 1.000 lời gọi API tổng cộng.

Dấu hiệu đầu tiên có gì đó sai xuất hiện lúc 11:23 sáng: dashboard theo dõi của tôi cho thấy quota cạn kiệt trên Tài khoản 1. Bình thường. Proxy nên chuyển sang Tài khoản 2.

Nó đã làm vậy. Tài khoản 2 cạn kiệt lúc 11:31 sáng.

Đến trưa, tất cả 15 tài khoản đều ở 100% quota. Proxy không còn tài khoản nào để failover. Nhưng đây là vấn đề thực sự: tôi không để ý. Các agent vẫn tiếp tục thử, proxy vẫn trả về lỗi, và job batch của tôi — được thiết kế để thử lại khi gặp lỗi — đang hammering vào endpoint, đốt hết phần token ít ỏi còn lại trên tất cả 15 tài khoản để cố lấy *bất kỳ* phản hồi nào.

Đến 2 giờ chiều, tất cả 15 tài khoản nhà cung cấp bị suspend.

Nguyên nhân: Job batch nặng token hơn nhiều so với tôi ước tính. Tôi đã không tính đến system prompt (mà tôi đã mở rộng đáng kể hai ngày trước đó), số lượng output token (Gemini tạo JSON dài dòng), hay việc chỉ riêng metadata ảnh đã tiêu thụ gấp 3 lần dự kiến.

Hiệu Ứng Dây Chuyền

Việc suspend tất cả 15 tài khoản không chỉ giết chết job batch. Nó giết chết mọi thứ chạy qua các tài khoản đó:

- Kiểm tra sức khỏe của gateway agent (chạy trên cron 3 phút, tiêu thụ token)
- Hai tiến trình agent chạy nền theo dõi server
- Content pipeline cho một dự án khác

Tôi dành sáu tiếng chiều thứ Năm để xử lý sự cố. Kháng cáo tài khoản với nhà cung cấp (hầu hết được khôi phục trong vòng 24 giờ). Hoàn thành thủ công job batch thông qua một API key khác. Viết lại cron kiểm tra sức khỏe để dùng model rẻ hơn.

Thay Đổi Sau Đó

Sự cố này buộc ra một tập quyết định mà lẽ ra tôi phải đưa ra nhiều tháng trước:

1. Theo dõi ngân sách token mỗi job

Trước khi bắt kỳ batch job nào chạy, tôi ước tính lượng token tiêu thụ mỗi yêu cầu:

```
# Ước tính sơ bộ trước khi khởi động batch
estimated_input_tokens = (
    system_prompt_tokens +      # Đo một lần, cache lại
    example_tokens +           # Few-shot examples nếu có
    request_content_tokens     # Nội dung mỗi item
)
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:,}")
```

Nếu ước tính vượt ngưỡng, tôi yêu cầu xác nhận rõ ràng trước khi tiến hành.

2. Circuit breaker trong logic retry

Job batch thử lại *mọi* lỗi, kể cả cạn kiệt quota. Đây là nguyên nhân trực tiếp của hiệu ứng dây chuyền. Sau sự cố này, mọi batch job đều có 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 check không dùng model đắt tiền

Mọi health check đang đốt token đều dùng cùng model với workload chính. Sau sự cố, health check chuyển sang dùng model rẻ nhất có thể (Gemini Flash, hoặc không dùng model nào cho các kiểm tra không cần AI). Tôi cũng xóa tất cả job health check dựa trên cron và thay bằng kiểm tra theo sự kiện, chỉ chạy khi có lưu lượng thực.

4. Phân vùng tài khoản

15 tài khoản nhà cung cấp nay được chia thành các pool: 10 cho workload production, 5 dành riêng cho health check và monitoring. Job batch không thể làm cạn pool monitoring.

5. Tối ưu file context

Trong phản ứng khẩn cấp, tôi kiểm tra mọi file context mà agent đang load. Kết quả đáng kinh ngạc:

File Context	Trước	Sau	Giảm
System prompt	4.200 token	380 token	91%
Định nghĩa tool	2.800 token	690 token	75%
Memory context	2.030 token	127 token	94%
Tổng	9.030 token	797 token	91%

Mỗi yêu cầu agent tiêu thụ 9.030 token *trước khi có bất kỳ nội dung thực nào*. Ở quy mô lớn, điều này rất tàn khốc. Công việc tối ưu — loại bỏ tài liệu dài dòng khỏi system prompt, nén định nghĩa tool, chuyển memory sang truy xuất theo yêu cầu — đã giảm chi phí baseline mỗi yêu cầu xuống 91%.

Chúng ta sẽ đề cập context engineering chi tiết trong Chương 7. Bản ngắn gọn: system prompt của bạn không phải file README. Viết nó như thể nó tốn tiền, vì nó tốn thật.

Nội Dung Cuốn Sách Đây

Cuốn sách này được tổ chức xung quanh vòng đời thực tế của AI agent trong production. Phần 1 đề cập nền tảng — agent thực sự là gì, cách chọn model và cách suy nghĩ về chi phí. Phần 2 đề cập kiến trúc — cách thiết kế hệ thống agent có thể quan sát được, đáng tin cậy và có thể phục hồi. Phần 3 đề cập những phần khó — điều phối đa agent, hệ thống bộ nhớ và context engineering. Phần 4 đề cập vận hành — monitoring, phản ứng sự cố, kiểm soát chi phí và bảo mật.

Mỗi chương bao gồm ít nhất một câu chuyện chiến trường từ hạ tầng thực tế của tôi. Đây không phải giả thuyết. Những sai lầm được mô tả là sai lầm tôi đã mắc, với chi phí thực (cả tài chính lẫn thứ khác). Các giải pháp được mô tả là những giải pháp đang chạy trong môi trường production của tôi.

Các ví dụ code là thật. Config là thật (với thông tin xác thực đã được xóa). Giá cả hiện tại tính đến tháng 3 năm 2026, dù các model thay đổi đủ thường xuyên để bạn nên xác minh trước khi đưa ra quyết định kiến trúc lớn.

Cuốn Sách Này Không Để Cập

Đây không phải cuốn sách về: - Xây dựng chatbot đầu tiên của bạn - Prompt engineering cho trường hợp sử dụng tiêu dùng - Cách dùng Claude hay ChatGPT từ giao diện web - Lý thuyết machine learning hay huấn luyện model - Xây dựng model tùy chỉnh

Nếu bạn hoàn toàn mới với AI API, bạn có thể muốn bắt đầu với tài nguyên giới thiệu hơn. Cuốn sách này giả định bạn đã ít nhất từng dùng AI API, hiểu các khái niệm HTTP cơ bản, và quen với terminal.

Cuốn Sách Này Dành Cho Ai

Đối tượng chính: Kỹ sư DevOps, SRE và platform engineer đang được yêu cầu chạy AI agent workload trong production. Người đã kế thừa một hệ thống agent người khác xây và tự hỏi tại sao nó cứ hỏng. Người chuẩn bị xây một hệ thống và muốn tránh các sai lầm hiển nhiên.

Đối tượng phụ: Kỹ sư backend xây dựng tính năng powered by agent cần hiểu các mối quan tâm về hạ tầng. Tech lead đưa ra quyết định kiến trúc về tích hợp AI. Engineering manager muốn hiểu những gì team họ đang nói về độ phức tạp của AI agent.

Không dành cho: Nhà nghiên cứu, học giả, hay người quan tâm chủ yếu đến khía cạnh lý thuyết của AI. Định hướng ở đây hoàn toàn thực tế: cái gì hoạt động, cái gì hỏng, cái gì tốn bao nhiêu, và cách sửa.

Một Ghi Chú Về Sự Thành Thật

Tôi sẽ nói thẳng về những thất bại xuyên suốt cuốn sách này. Hóa đơn Gemini \$500 không phải sự cố xấu hổ duy nhất bạn sẽ đọc. Có những chương mô tả hệ thống đa agent bị deadlock, agent chỉnh sửa nhầm config và làm hỏng dịch vụ production, và một sự cố bảo mật liên quan đến token bị lộ qua lịch sử chat.

Điều này là có chủ ý. Cộng đồng DevOps có truyền thống mạnh về postmortem không đổ lỗi và báo cáo sự cố trung thực vì chúng tôi đã học — thường rất đau — rằng giả vờ thất bại không xảy ra ngăn cản việc học hỏi. Tôi đang áp dụng truyền thống đó vào hạ tầng AI agent.

Ngành AI, ngược lại, có truyền thống mạnh về việc chỉ trình diễn những demo hoạt động. Cuốn sách này là mặt còn lại của đó.

Cách Sử Dụng Cuốn Sách Này

Nếu bạn đang đánh giá AI agent cho production: Đọc Chương 1-3 (Phần 1) và Chương 12 (Kiểm Soát Chi Phí). Đây là khung tư duy và con số bạn cần để đánh giá.

Nếu bạn đang thiết kế hệ thống agent: Đọc Chương 4-8 (Phần 1-2). Các chương kiến trúc sẽ cứu bạn khỏi những sai lầm cấu trúc phổ biến nhất.

Nếu bạn đang debug hệ thống agent hiện tại: Chuyển thẳng đến Chương 14 (Câu Chuyện Chiến Trường Debug). Sau đó đọc các chương nền tảng liên quan đến vấn đề cụ thể của bạn.

Nếu bạn đang nhận cảnh báo về agent trong production: Chương 13 (Phản Ứng Sự Cố cho Hệ Thống AI) trước. Phần còn lại có thể chờ đến sáng.

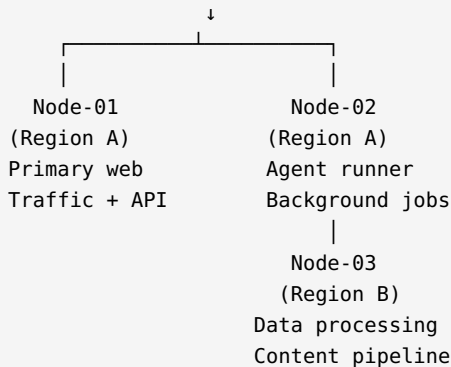
Nếu bạn muốn hiểu toàn bộ bức tranh: Đọc theo thứ tự. Có tham chiếu chéo, nhưng mỗi chương được thiết kế để đứng độc lập nếu cần.

Thiết Lập Khiến Mọi Thứ Trở Nên Thực Tế

Xuyên suốt cuốn sách, tôi sẽ tham chiếu đến hạ tầng thực tế của chúng tôi. Đây là bản đồ nhanh để bạn định hướng khi tôi đề cập đến các thành phần cụ thể:

Infrastructure Map (March 2026)

Internet → Load Balancer (cloud provider)



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)

Khi tôi nói “agent đã SSH vào Node-01,” đây là điều tôi muốn nói. Khi tôi nói “proxy đã xoay sang tài khoản nhà cung cấp tiếp theo,” đây là hệ thống đang thực hiện việc xoay.

Trước Khi Bắt Đầu

Tôi muốn làm rõ một điều trước khi đi tiếp: AI agent trong production thực sự mạnh mẽ. Lý do tôi vận hành hạ tầng này không phải là khổ dâm. Gateway agent của chúng tôi đã tiết kiệm hàng giờ mỗi tuần. Content pipeline tạo ra trong vài giờ thứ sẽ mất nhiều ngày nếu làm thủ công. Claude Code, chạy như agent với quyền truy cập filesystem, đã viết, test và deploy tính năng trong khi tôi ngủ.

Tiền đề của cuốn sách này không phải “AI agent nguy hiểm, hãy tránh.” Tiền đề là “AI agent là công cụ mạnh mẽ đòi hỏi sự chuyên nghiệp như bất kỳ hạ tầng production nào khác.”

Bạn sẽ không deploy database mới mà không có monitoring, backup và kế hoạch phản ứng sự cố. Bạn sẽ không chạy job queue mà không có dead letter queue, giới hạn retry và circuit breaker. AI agent xứng đáng được đối xử với sự nghiêm túc tương tự.

Các câu chuyện chiến trường trong cuốn sách này không phải là lý lẽ chống lại việc xây dựng với AI agent. Chúng là lý lẽ để xây dựng với AI agent *đúng cách*.

Hãy bắt đầu thôi.

Điểm Mấu Chốt

- AI agent trong production có khoảng cách giữa hứa hẹn của demo và thực tế vận hành mà cuốn sách này giải quyết trực tiếp
 - Hạ tầng agent production thực sự cần monitoring, kiểm soát chi phí, circuit breaker và phản ứng sự cố — giống như bất kỳ hệ thống production nào
 - Sự cố Gemini ~\$500 và 15 tài khoản bị suspend đều do thiếu guardrail, không phải do lỗi model
 - Context engineering (kiểm soát những gì đi vào mỗi yêu cầu) là một trong những tối ưu có đòn bẩy cao nhất — giảm 91% token là có thể đạt được
 - Bức tranh AI thay đổi đáng kể từ 2024→2026: model cải thiện vượt bậc, nhưng thách thức vận hành vẫn còn
 - Cuốn sách này được viết từ góc nhìn DevOps/SRE: uptime, chi phí, blast radius và khả năng nhận cảnh báo lúc 3 giờ sáng
-

Tiếp theo: Chương 2 — AI Agent Thực Sự Là Gì?

Chương 2: AI Agent Thực Sự Là Gì?

“Agent chỉ là một model trong vòng lặp với quyền truy cập tool. Phần đáng sợ là điều đó quan trọng đến mức nào.” — Kỹ sư cấp cao, trong buổi review sau sự cố

Vấn Đề Định Nghĩa

Hỏi mười người “AI agent” là gì và bạn sẽ nhận được mười hai định nghĩa. Đội marketing dùng nó để chỉ bất kỳ tính năng AI nào. Nhà nghiên cứu dùng nó cho hệ thống có tính chất tự chủ chính thức. Nhà cung cấp dùng nó cho bất cứ thứ gì họ đang bán trong quý này.

Cho mục đích của cuốn sách này, đây là định nghĩa làm việc:

AI agent là hệ thống dùng language model để quyết định hành động nào cần thực hiện, sau đó thực hiện những hành động đó, và tiếp tục chu kỳ này cho đến khi mục tiêu đạt được hoặc điều kiện kết thúc được đáp ứng.

Thế thôi. Không có phép màu. Không có ý thức. Không có tri giác. Một model trong vòng lặp với tool và điều kiện dừng.

Định nghĩa này cố tình không hào nhoáng, vì các thách thức kỹ thuật của AI agent cũng không hào nhoáng. Chúng là những thách thức tồn tại trong bất kỳ hệ thống tự động nào: quản lý trạng thái, phục hồi lỗi, observability, kiểm soát chi phí, và câu hỏi rất nguời này: “điều gì xảy ra khi nó làm điều tôi không mong đợi?”

Hãy xây dựng từ các nguyên tắc cơ bản.

Language Model Thực Sự Là Gì

Trước khi nói về agent, chúng ta cần làm rõ những gì language model làm, vì nhiều lỗi agent bắt nguồn từ hiểu nhầm điều này.

Language model là một hàm: **cho một chuỗi token (văn bản), dự đoán phân phối xác suất trên token tiếp theo.**

Đó là toàn bộ cơ chế. Mọi thứ khác — sử dụng tool, suy luận, tạo code, tuân theo hướng dẫn — đều xuất hiện từ việc huấn luyện hàm này trên lượng văn bản do con người tạo ra khổng lồ, sau đó fine-tune để tuân theo hướng dẫn và dùng các tool được cung cấp.

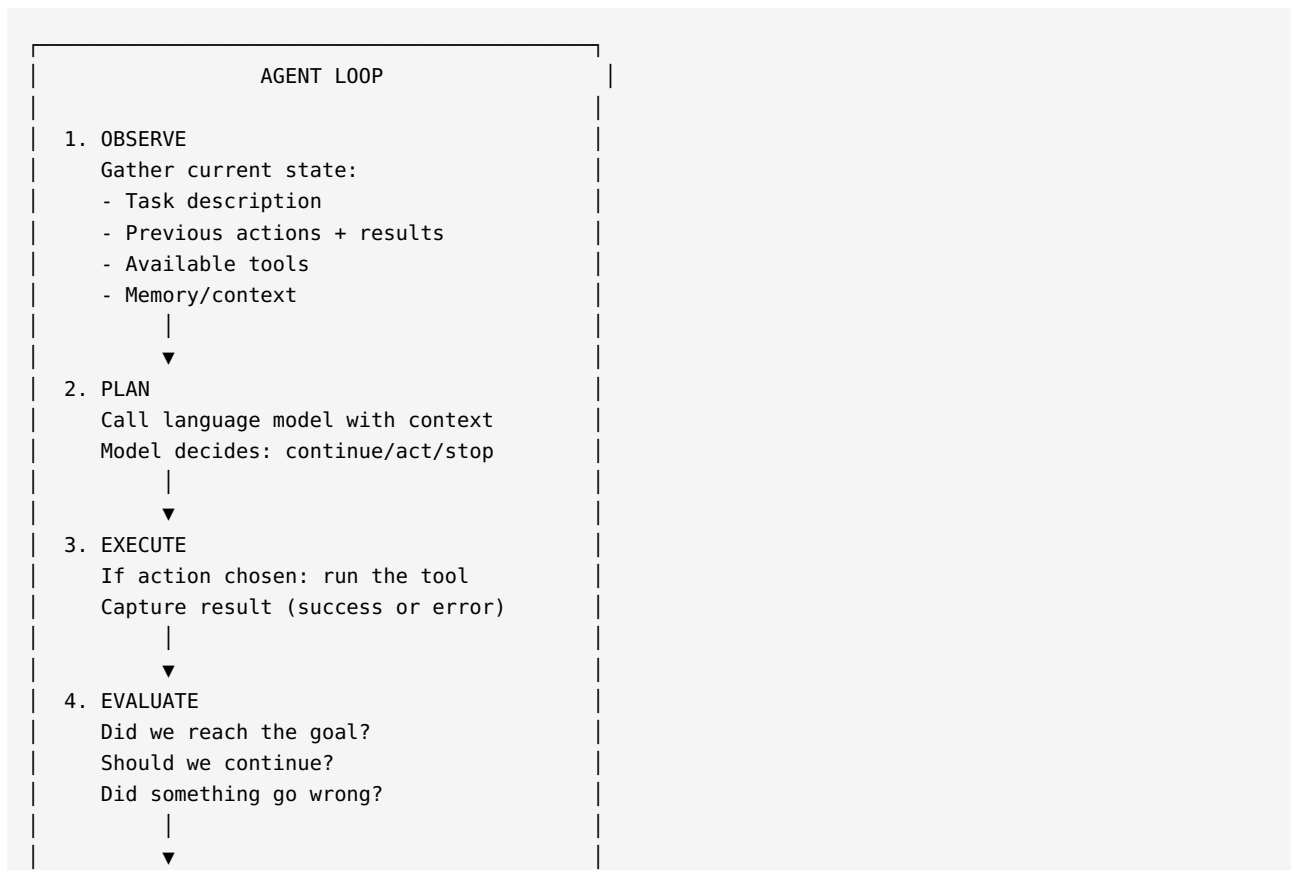
Điều này có một hệ quả quan trọng: **language model không có trạng thái giữa các lần gọi**. Mỗi lời gọi API là stateless. Model không nhớ lần gọi trước. Nó chỉ thấy những gì bạn đưa vào context window của yêu cầu hiện tại.

Khi chúng ta nói về agent có “bộ nhớ”, chúng ta đang nói về hệ thống đưa thông tin quá khứ liên quan vào mỗi yêu cầu mới. Bộ nhớ nằm trong code, không phải trong model.

Đây không phải giới hạn cần khắc phục. Đây là nguyên tắc thiết kế cần nội tâm hóa. Khi bạn hiểu rằng bộ nhớ là lựa chọn kỹ thuật, không phải tính năng của model, bạn bắt đầu đưa ra quyết định tốt hơn về những gì cần nhớ, cách cấu trúc nó, và khi nào thì loại bỏ.

Vòng Lặp Agent

Mọi AI agent, bất kể framework hay độ phức tạp, đều thực thi một biến thể của vòng lặp này:



```
Loop back to OBSERVE
(or terminate)
```

Vòng lặp này chạy cho đến khi một trong các điều kiện sau: - Model quyết định task đã hoàn thành - Điều kiện kết thúc được đáp ứng (giới hạn bước, giới hạn thời gian, giới hạn ngân sách) - Xảy ra lỗi không thể phục hồi - Tín hiệu bên ngoài dừng vòng lặp (con người trong vòng lặp, circuit breaker)

Đây là vòng lặp tương tự trong Python — một triển khai agent tối giản nhưng hoạt động được:

```
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}

```

Đây là khung xương của mọi hệ thống agent tôi chạy trong production, được mở rộng với: - Đếm token và áp dụng ngân sách trước mỗi bước - Ghi log có cấu trúc mọi lần gọi model và thực thi tool - Xử lý timeout ở cấp tool - Circuit breaker cho các lỗi lặp lại - Trạng thái lưu trữ để các lần chạy có thể tiếp tục sau khi bị gián đoạn

Chúng ta sẽ đề cập tất cả những điều này ở các chương sau. Bây giờ hãy nội tâm hóa vòng lặp: **Quan sát → Lập kế hoạch → Thực thi → Đánh giá → lặp lại.**

Điều Phân Biệt Agent Với Chatbot

Sự phân biệt quan trọng về mặt vận hành, không phải triết học.

Thuộc tính	Chatbot	Copilot	Agent
Lượt hội thoại	Do người dùng điều khiển	Do người dùng điều khiển	Tự điều khiển
Quyền truy cập tool	Tối thiểu hoặc không có	Chỉ đọc	Đọc và ghi
Tác động phụ	Không có	Chỉ gợi ý	Thay đổi trực tiếp
Thời gian	Một lượt	Một lượt	Nhiều lượt, tự động
Con người trong vòng lặp	Mỗi lượt	Mỗi lượt	Tùy chọn hoặc không
Hồ sơ chi phí	Mỗi truy vấn	Mỗi truy vấn	Tích lũy
Blast radius khi lỗi	Thấp	Thấp	Có thể cao
Nhu cầu observability	Thấp	Thấp	Cao

Chatbot chờ bạn nói gì đó. Agent tự quyết định phải làm gì tiếp theo.

Copilot (GitHub Copilot, Claude trong trình soạn thảo) gợi ý. Bạn chấp nhận hoặc từ chối. Nó không thể thay đổi file mà không có sự chấp thuận của bạn.

Agent thực hiện hành động. Nó có thể đọc file, ghi file, chạy lệnh shell, gọi API, chỉnh sửa database — bất cứ tool nào bạn cấp. Nếu bạn cấp cho agent quyền ghi vào database production và nó đưa ra quyết định sai, database sẽ sai. Không có lời nhắc “bạn có chắc không?” trừ khi bạn tự xây dựng.

Đây là lý do tại sao kỷ luật kỹ thuật xung quanh agent quan trọng hơn nhiều so với chatbot. **Blast radius của một lỗi agent tỷ lệ thuận với quyền hạn bạn cấp cho nó.**

Các Loại Kiến Trúc Agent

Agent Đơn

Kiến trúc đơn giản nhất: một model, một vòng lặp, một bộ tool.

User → Agent (Model + Tools) → Result

Khi nào dùng: Task có phạm vi rõ ràng. Một agent đơn quản lý deployment, tóm tắt log, hoặc tạo nội dung cho một pipeline cụ thể. Agent làm một loại công việc và làm tốt.

Dạng lỗi: Điểm lỗi đơn. Nếu model bị nhầm lẫn, không có ý kiến thứ hai. Task dài tích lũy context cho đến khi window đầy, đòi hỏi quản lý context cẩn thận.

Cách chúng tôi dùng: Tạo nội dung mỗi bài blog. Agent nhận tiêu đề và outline, tạo nội dung, ghi vào file. Đầu vào rõ ràng, đầu ra rõ ràng, phạm vi hữu hạn.

Đa Agent (Orchestrator + Specialist)

Một orchestrator agent phân rã task phức tạp và ủy thác cho các subagent chuyên biệt.

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

Khi nào dùng: Task tự nhiên phân rã thành các giai đoạn riêng biệt. Khi bạn muốn chuyên gia — một agent tối ưu hóa nghiên cứu, agent chuyên code, review agent với tiêu chí khác.

Dạng lỗi: Lỗi orchestrator gây hiệu ứng dây chuyền. Giao tiếp giữa các agent đòi hỏi thiết kế interface cẩn thận (orchestrator truyền định dạng nào cho writer? Writer trả về gì?). Debug khó hơn vì lỗi có thể phát sinh ở bất kỳ cấp nào.

Cách chúng tôi dùng: Cặp đôi Claude Code + gateway agent. Claude Code xử lý triển khai; gateway agent xử lý tự động hóa trình duyệt, nhắn tin và các task cần xác nhận trực quan. Chúng giao tiếp qua thư mục workspace chia sẻ và hệ thống nhắn tin.

Mesh / Swarm

Nhiều agent hoạt động song song, có thể theo dõi nhau, không có orchestrator đơn.

```
Agent A ↔ Agent B
  †           †
Agent C ↔ Agent D
```

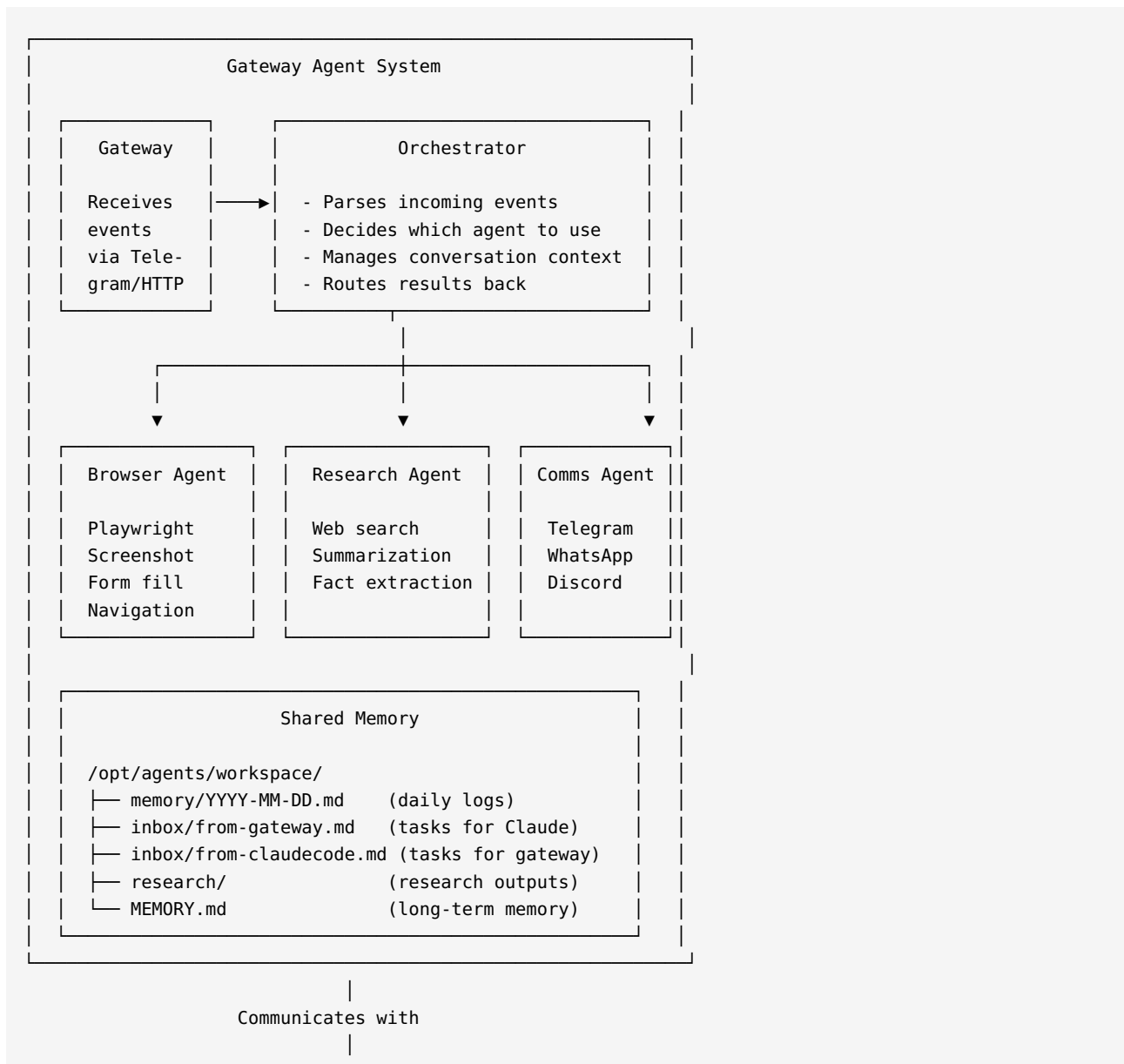
Khi nào dùng: Task throughput cao nơi tính song song quan trọng hơn điều phối. Task monitoring nơi sự dư thừa có giá trị.

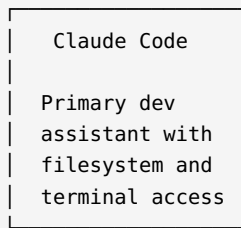
Dạng lỗi: Phức tạp. Race condition. Công việc trùng lặp. Chi phí điều phối có thể vượt quá lợi ích của tính song song. Chi phí token nhân lên với mỗi agent.

Kinh nghiệm của chúng tôi: Tôi đã chạy hệ thống monitoring mesh vào cuối năm 2025. Sáu agent, mỗi agent chịu trách nhiệm một tập hợp con dịch vụ, báo cáo vào state store chia sẻ. Monitoring hoạt động. Các agent cũng chạy health check mỗi 3 phút mỗi agent, tiêu thụ token liên tục. Chi phí đủ lớn để tôi tắt nó và thay bằng event-driven monitoring. Thêm về điều này trong phần câu chuyện chiến trường.

Kiến Trúc Gateway Agent: Hệ Thống Đa Agent Thực Tế

Thay vì bịa ra một ví dụ giả thuyết, hãy để tôi mô tả kiến trúc đa agent thực sự đang chạy trên hạ tầng của chúng tôi tính đến tháng 3 năm 2026.





Các quyết định thiết kế quan trọng trong kiến trúc này:

1. Phân tách mối quan tâm: Mỗi subagent có phạm vi hẹp. Browser agent không biết về Telegram. Research agent không có quyền ghi filesystem. Phạm vi hẹp giới hạn blast radius và làm cho việc debug có thể thực hiện được.

2. Giao tiếp qua file: Các agent giao tiếp qua các file trong `/opt/agents/workspace/`, không phải lời gọi API trực tiếp với nhau. Điều này có nghĩa giao tiếp là: - Bền vững (tồn tại sau khi khởi động lại) - Có thể kiểm tra (tôi có thể đọc thư mục inbox để xem những gì đã gửi) - Tách rời (các agent không cần chạy đồng thời) - Có thể debug (tôi có thể viết thủ công vào inbox để test hành vi)

3. Quản lý bộ nhớ rõ ràng: Bộ nhớ không chỉ là lịch sử hội thoại tích lũy. Gateway agent ghi nhật ký hàng ngày có cấu trúc, duy trì file MEMORY.md dài hạn, và truyền context liên quan cho subagent thay vì toàn bộ lịch sử. Điều này giúp context window quản lý được.

4. Hướng sự kiện, không phải polling: Gateway agent không chạy theo lịch trình, kiểm tra mọi thứ mỗi N phút. Nó phản hồi các sự kiện (tin nhắn Telegram, thông báo từ Claude Code, sự kiện hệ thống). Đây là bài học từ thí nghiệm mesh monitor.

Đây là vòng lặp xử lý sự kiện đơn giản hóa trong orchestrator:

```
class GatewayOrchestrator:
    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("/opt/agents/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)

```

Model định tuyến luôn là option rẻ nhất có thể làm việc một cách đáng tin cậy. Đây là chủ đề tái diễn: khớp khả năng model với độ phức tạp task. Quyết định định tuyến cần phân loại, không cần suy luận. Dùng Haiku cho phân loại. Để dành Opus cho những task thực sự cần nó.

Tool: Đôi Tay Của Agent

Agent không có tool chỉ là chatbot suy nghĩ nhiều hơn. Tool là thứ cho agent khả năng tác động đến thế giới.

Thiết kế tool là một trong những khía cạnh quan trọng nhất và bị đánh giá thấp nhất của phát triển agent. Tool thiết kế kém dẫn đến: - Agent gọi tool với đối số sai (interface không rõ ràng) - Lỗi không thể phục hồi (tool không idempotent không có biện pháp bảo vệ) - Lỗi hỏng bảo mật (tool với quyền quá mức) - Chi phí đột biến (tool tự tiêu thụ token hoặc tiền)

Nguyên Tắc Thiết Kế Tool

1. Làm tool atomic và tập trung

Tool làm một việc để sử dụng đúng hơn tool làm nhiều việc. Thay vì:

```
# 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'."""
    ...
```

Ưu tiên:

```
# 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."""
    ...
```

Model phải quyết định tool nào cần gọi. Hãy cho nó những lựa chọn đơn giản, không mơ hồ.

2. Trả về kết quả có cấu trúc, đầy đủ thông tin

Kết quả tool là “quan sát” tiếp theo của model. Làm nó đủ phong phú để model hiểu điều gì đã xảy ra.

```
# 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)
    except:
        return {"status": "error", "message": str(result)}
    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. Xây dựng biện pháp bảo vệ vào tool nguy hiểm

Tool có tác động phụ phá hủy nên bao gồm cơ chế xác nhận hoặc giới hạn:

```

@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+~\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. Làm tool có thể quan sát

Mỗi lần gọi tool nên được ghi log. Bạn cần biết agent của bạn đã làm gì.

```

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
status={result.get('status', 'unknown')}")
            return result
        except Exception as e:
            duration = time.time() - start
            logger.error(f"TOOL_ERROR [{call_id}] {func.__name__}: duration={duration:.2f}s
error={e}")
            raise
        return wrapper

```

Câu Chuyện Chiến Trường: Hệ Thống Agent Mesh Monitor Tự Đốt Mình

Quý 4 năm 2025, tôi đã xây dựng thứ có vẻ là giải pháp thanh lịch cho monitoring hạ tầng.

Thiết Lập

Sáu agent nhẹ, mỗi agent được giao theo dõi một tập hợp con dịch vụ: - Agent Alpha: Web server (Node-01, Node-02) - Agent Beta: Database (PostgreSQL + Redis) - Agent Gamma: Dịch vụ API (API proxy, application endpoint) - Agent Delta: Content delivery và static asset - Agent Epsilon: Chứng chỉ SSL và DNS - Agent Zeta: Độ sâu hàng đợi workload AI

Mỗi agent chạy theo cron 3 phút, kiểm tra dịch vụ được giao và ghi trạng thái vào file trạng thái chia sẻ. Agent “coordinator” thứ bảy đọc file trạng thái mỗi 10 phút và gửi tóm tắt hàng ngày lên Telegram.

Thiết kế thực sự vững chắc từ góc độ độ tin cậy. Các agent độc lập — lỗi ở Agent Alpha không ảnh hưởng Agent Beta. Trạng thái chia sẻ là một file JSON đơn giản với file locking để tránh ghi đồng thời. Không agent nào có quyền ghi vào bất cứ đâu ngoại trừ file trạng thái.

Vấn Đề

Tiêu thụ token ở quy mô lớn.

Mỗi lần gọi agent, bao gồm system prompt, định nghĩa tool và bản kiểm tra monitoring, tiêu thụ xấp xỉ:

- System prompt: 380 token
- Định nghĩa tool (6 monitoring tool): 890 token
- Mô tả task: ~120 token
- Phản hồi: ~200-400 token
- Thực thi tool + kết quả: ~150-300 token

Tổng mỗi lần kiểm tra: khoảng **1.700 token mỗi lần chạy agent.**

Sáu agent, mỗi 3 phút, bằng 2 lần chạy agent mỗi phút trên toàn fleet, 2.880 lần chạy mỗi ngày. Với 1.700 token mỗi lần chạy:

```
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
```

Coordinator agent thêm khoảng \$3/ngày nữa.

\$37,56 mỗi ngày để biết server có đang chạy không.

Để so sánh: chi phí server thực tế của tôi là \$48/tháng. Tôi đang chi nhiều hơn cho monitoring so với tính toán.

Vấn Đề Tệ Hơn

Các agent phát hiện vấn đề. Tôi đã xây phần đó đúng. Điều tôi không lường trước là *các agent phát hiện vấn đề cũng tạo ra nhiều token hơn*. Kiểm tra bình thường: 1.700 token. Kiểm tra phát hiện vấn đề và cần điều tra thêm: 4.000-8.000 token, vì agent sẽ gọi thêm tool chẩn đoán, xem log, tham chiếu chéo với trạng thái dịch vụ khác.

Vào ngày hạ tầng tệ — giả sử, chúng chỉ sắp hết hạn, hai dịch vụ có tỷ lệ lỗi cao, và connection pool database đang nóng — các monitoring agent sẽ mỗi agent đi sâu hơn, tạo ra 5 lần tiêu thụ token bình thường. Những ngày tôi cần monitoring nhất là những ngày monitoring tốn nhiều nhất.

Quyết Định

Sau ba tuần, tôi tắt toàn bộ hệ thống mesh monitoring và thay bằng:

1. Push-based alerting: Dịch vụ đẩy cảnh báo đến webhook khi chúng phát hiện vấn đề của chính mình. Không polling. Không liên quan agent cho đến khi có cảnh báo đáng điều tra.

2. Agent coordinator đơn, hướng sự kiện: Coordinator chỉ chạy khi cảnh báo đến. Nó nhận cảnh báo, gọi các tool chẩn đoán liên quan và gửi tóm tắt có cấu trúc. Một agent, theo yêu cầu, thay vì sáu agent theo lịch trình.

3. Kiểm tra HTTP đơn giản: Script nhẹ (không AI, không gọi model) poll endpoint `/health` mỗi 30 giây và kích hoạt cảnh báo khi lỗi. 0 token tiêu thụ. Cron job chạy lệnh `curl` không tốn gì.

4. AI agent cho chẩn đoán, không phải phát hiện: Khi cảnh báo kích hoạt, *lúc đó* mới đưa AI agent vào để chẩn đoán. Agent kiểm tra log, tham chiếu chéo dịch vụ và đề xuất khắc phục. Đây là cách dùng tốt của khả năng agent — suy luận phức tạp về dữ liệu mơ hồ. Phát hiện là cách dùng xấu — nó là khớp mẫu trên trạng thái đã biết, tốt hơn khi dùng tool monitoring chuyên dụng.

Hệ thống sửa đổi tốn khoảng \$0.50/ngày cho AI token — cho các lần chạy chẩn đoán đôi khi xảy ra khi có gì đó thực sự sai.

Bài Học

Dùng AI agent cho các task đòi hỏi trí thông minh. Dùng tool chuyên dụng cho các task đòi hỏi độ tin cậy và khả năng lặp lại.

Health check không phải task thông minh. Kiểm tra xem endpoint HTTP có trả về 200 không là thứ không được lợi từ language model. Nó được lợi từ chương trình đơn giản, đáng tin cậy, rẻ tiền làm chính xác điều đó và không có gì khác.

AI agent tỏa sáng khi: - Task có đầu vào mơ hồ cần giải thích - Con đường đến hoàn thành chưa biết trước - Nhiều nguồn thông tin cần được tổng hợp - Vấn đề đòi hỏi suy luận trong điều kiện không chắc chắn

AI agent đắt tiền và quá mức cần thiết khi: - Task được định nghĩa rõ ràng và xác định - Cùng một logic chạy theo lịch trình bất kể context - Tốc độ và chi phí quan trọng hơn trí thông minh - Khớp mẫu đơn giản là đủ

Mesh monitor là cách tốn kém để học bài học này. Tôi sẽ không mắc sai lầm đó nữa — và tôi đang truyền bài học sớm cho bạn để bạn không phải trải qua.

Agent vs. Chatbot vs. Copilot: Bảng Thực Tế

Đây là bảng so sánh tôi thực sự dùng khi quyết định loại tool nào cần sử dụng:

Câu hỏi	Chatbot	Copilot	Agent
Cần thực hiện hành động không?	Không	Chỉ gợi ý	Có
Cần chạy tự động không?	Không	Không	Có
Có người xem xét từng bước không?	Có	Có	Tùy chọn
Có quyền ghi vào hệ thống không?	Không	Hiếm khi	Thường
Có chạy khi tôi ngủ không?	Không	Không	Có
Điều gì xảy ra khi mắc lỗi?	Diễn đạt lại và thử lại	Từ chối gợi ý	Có thể xấu
Cần audit log không?	Tốt nếu có	Tốt nếu có	Bắt buộc
Cần kiểm soát chi phí không?	Tối thiểu	Tối thiểu	Quan trọng

Nếu bạn đang xây dựng gì đó và những câu trả lời trong cột “Agent” làm bạn sợ, tốt. Nỗi sợ đó phù hợp và nên thúc đẩy kỹ thuật đúng đắn. Nếu câu trả lời trong cột “Agent” không làm bạn sợ, hãy đọc lại câu hỏi về “điều gì xảy ra khi mắc lỗi.”

Phiên Bản Node.js: Cho Kỹ Sư Server-Side

Cho team xây dựng agent bằng Node.js/TypeScript, đây là vòng lặp agent tương đương:

```
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,
    };
  }
};

```

```
}  
}
```

Kết Thúc: Điều Quan Trọng Nhất Bạn Chưa Nghĩ Đến

Mọi hệ thống agent production đều cần logic kết thúc rõ ràng. “Model quyết định khi nào dừng” là không đủ.

Model có thể rơi vào vòng lặp. Model có thể quyết định cần làm nhiều hơn nhiều so với task yêu cầu. Model có thể ảo giác kết quả tool và tiếp tục dựa trên niềm tin sai. Bất kỳ điều nào trong số này có thể chạy cho đến khi bạn chạm giới hạn context, ngân sách token, hoặc giới hạn tín dụng.

Kiểm soát kết thúc tối thiểu cho bất kỳ agent production nào:

```
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})"  
)
```

Đặt giới hạn chặt hơn bạn nghĩ cần. Bạn luôn có thể nói lỏng. Bạn không thể hoàn trả token đã tiêu.

Điểm Mấu Chốt

- AI agent là language model trong vòng lặp với tool — vòng lặp chạy cho đến khi mục tiêu đạt được hoặc giới hạn được đáp ứng
 - Vòng lặp agent: **Quan sát** → **Lập kế hoạch** → **Thực thi** → **Đánh giá** → **lặp lại**
 - Language model là stateless giữa các lần gọi; “bộ nhớ” là lựa chọn kỹ thuật, không phải tính năng model
 - Ba kiến trúc: agent đơn (đơn giản, tập trung), đa agent orchestrator+specialist (task phức tạp), mesh/swarm (song song throughput cao)
 - Thiết kế tool là quan trọng: atomic, có kiểu dữ liệu rõ ràng, với biện pháp bảo vệ cho thao tác phá hủy, và có thể quan sát qua logging
 - Câu chuyện chiến trường mesh monitoring minh họa trường hợp lạm dụng agent phổ biến nhất: dùng AI đắt tiền cho task tốt hơn khi dùng tool rẻ, chuyên dụng
 - **Luôn triển khai giới hạn kết thúc**: max step, max duration, max token, max cost. “Model quyết định khi nào dừng” không an toàn cho production
-

Tiếp theo: Chương 3 — Chọn Model Của Bạn: Hướng Dẫn Của Kỹ Sư DevOps

Chương 3: Chọn Model Của Bạn: Hướng Dẫn Của Kỹ Sư DevOps

“Chọn sai model không phải lỗi kỹ thuật. Đó là lỗi ngân sách.” — Bài học rút ra sau khi định tuyến task debug hạ tầng đến Gemini Flash

Vấn Đề Lựa Chọn Model

Năm 2024, việc chọn model tương đối đơn giản. Có hai lựa chọn nghiêm túc: GPT-4 hoặc Claude 3. Bạn chọn một dựa trên cảm giác, độ tin cậy API, và liệu công ty bạn có mối quan hệ nhà cung cấp sẵn có hay không.

Tháng 3 năm 2026, bạn có khoảng bốn mươi model cấp production trên sáu nhà cung cấp, với hồ sơ khả năng, mức giá, kích thước context window và dạng lỗi khác nhau có ý nghĩa. Lựa chọn sai tốn tiền thật. Lựa chọn đúng có thể giảm chi phí inference xuống 90% mà không giảm chất lượng.

Chương này cung cấp khung quyết định tôi dùng cho hạ tầng của mình. Nó có quan điểm rõ ràng, dựa trên workload thực tế, và thành thật về điểm mạnh và điểm yếu của từng dòng model.

Bức Tranh Giá Cả (Tháng 3 Năm 2026)

Trước quyết định, hãy xem dữ liệu. Tất cả giá theo mỗi triệu token (MTok), input/output tương ứng.

Claude 4.x — Anthropic

Model	Input	Output	Context	Ghi chú
Opus 4.6	\$5	\$25	200K (1M beta)	Suy luận tốt nhất, bắt buộc cho agent phức tạp
Sonnet 4.6	\$3	\$15	200K	Cân bằng tốc độ/thông minh tốt nhất
Haiku 4.5	\$1	\$5	200K	

Model	Input	Output	Context	Ghi chú
				Nhanh nhất, throughput cao nhất

Cấu trúc giá của Anthropic thường cho lựa chọn đúng theo task. Opus đắt gấp 5 lần input so với Haiku. Nếu bạn dùng Opus cho task mà Haiku có thể xử lý, bạn đang đốt 5 lần ngân sách mà không tăng chất lượng.

Lưu ý quan trọng về giá context: Không giống một số nhà cung cấp, Anthropic hiện không tính phụ phí cho yêu cầu context dài trong window tiêu chuẩn. Prompt caching (beta) có thể giảm chi phí context lặp lại lên đến 90% — quan trọng cho workload agent nơi system prompt đến mỗi yêu cầu.

Dòng GPT — OpenAI

Model	Input	Output	Context	Ghi chú
GPT-4o	\$2,50	\$10	128K	Đa năng mạnh, hệ sinh thái rộng
GPT-4.1	\$2	\$8	1M context	Không phụ phí context dài — giá trị tốt nhất ở quy mô lớn
o3	\$10	\$40	200K	Model suy luận, đắt tiền
o4-mini	\$1,10	\$4,40	200K	Rẻ hơn o3 80%, suy luận tốt
GPT-5.3-Codex	Xem OpenAI	Xem OpenAI	200K	Chuyên code, mạnh về triển khai

GPT-4.1 của OpenAI đặc biệt thú vị cho workload agent cần context lớn mà không trả phụ phí nhân 2 lần như Gemini. Context window 1M ở \$2/MTok input thực sự cạnh tranh.

o3 và o4-mini là model suy luận — chúng dành nhiều tính toán hơn để “suy nghĩ” trước khi phản hồi. Dùng chúng cho vấn đề được lợi từ suy xét kéo dài: lập kế hoạch phức tạp, suy luận toán học, logic nhiều bước. Dùng dùng cho task đơn giản; bạn trả tiền cho suy luận dù task có cần hay không.

Gemini – Google

Model	Input	Output	Context	Ghi chú
2.5 Pro	\$1,25	\$10	1M	Phụ phí context dài: 2x trên 200K token
2.5 Flash	\$0,30	\$2,50	1M	Tỷ lệ chi phí/hiệu suất tốt nhất hiện có
2.5 Flash-Lite	~\$0,10	~\$0,40	1M	Cực rẻ, task đơn giản hơn

Hệ số nhân 2x của Gemini trên 200K token là chi tiết quan trọng cho workload agent. Nếu agent của bạn tích lũy context qua nhiều lượt — lịch sử hội thoại, kết quả tool, bộ nhớ — bạn có thể vượt 200K nhanh chóng. Yêu cầu 300K token tốn gấp đôi so với ước tính từ mức cơ bản.

Gemini Flash là vô địch giá-hiệu suất cho workload khối lượng cao nơi yêu cầu thông minh ở mức vừa phải. Phân loại nội dung, định dạng, trích xuất, tạo đơn giản: Flash xử lý tất cả với 1/10 chi phí của Opus.

Sự cố ~\$500 tôi đề cập ở Chương 1 dùng Gemini 1.5 Pro. Sự kết hợp của output dài dòng, context dài, hệ số nhân 2x và vòng lặp đột biến đã tạo ra hóa đơn đó. Flash sẽ rẻ hơn và — cho task cụ thể đó (tạo metadata ảnh) — có khả năng tương đương.

Mã Nguồn Mở / Tự Lưu Trữ / Bên Thứ Ba

Model	Input	Output	Context	Ghi chú
DeepSeek R1	\$0,55	\$2,19	128K	MoE 671B, suy luận mạnh
DeepSeek V3.2	\$0,14	\$0,28	64K	Cực rẻ, khả năng đáng ngạc nhiên
Llama 4 Maverick	\$0,27	\$0,85	1M	Meta, context lớn, chất lượng cạnh tranh
Mistral Nemo	\$0,02	\$0,04	128K	Siêu nhẹ, chỉ task đơn giản

DeepSeek và Llama đã trưởng thành đáng kể từ năm 2024. Cho workload nhạy cảm chi phí nơi bạn có thể chịu đựng độ tin cậy thấp hơn một chút và đôi khi cần debug các quirk đặc thù của model, đây là khoản tiết kiệm thực sự.

Lưu ý thực tế: các model này có sẵn qua các nhà cung cấp như Together AI, Groq và Fireworks AI. Các interface API thường tương thích OpenAI, nghĩa là code hiện tại của bạn có thể cần ít thay đổi để chuyển đổi nhà cung cấp. Nhưng hành vi không giống nhau — test kỹ trước khi chuyển workload production.

Cách Suy Nghĩ Về Lựa Chọn Model

Sai lầm hầu hết mọi người mắc là coi lựa chọn model như câu hỏi về chất lượng: “model nào thông minh nhất?”

Câu hỏi đúng là: “khả năng tối thiểu cần thiết để hoàn thành task này một cách đáng tin cậy là gì, và đâu là model rẻ nhất đáp ứng ngưỡng đó?”

Thang Khả Năng

Hãy nghĩ về model như thang bậc nơi mỗi bậc tốn hơn nhưng cung cấp nhiều khả năng hơn:

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 — GPT-4.1	— Balanced tasks, code review, content generation, multi-tool agents where reliability matters
\$10/MTok out	— GPT-4o — Gemini 2.5 Pro	— Strong general tasks, wide tool ecosystem, familiar to many teams
\$5/MTok out	— Claude Haiku 4.5 — o4-mini	— High-throughput, routing, classification, simple agents, fast response needed
\$2.50/MTok	— Gemini 2.5 Flash — DeepSeek R1	— Volume tasks, content processing, extraction, formatting
\$0.28/MTok	— DeepSeek V3.2 — Llama 4 Maverick	— Ultra-cheap, simple tasks, drafts, non-critical classification
\$0.04/MTok	— Mistral Nemo	— Trivial tasks only. Evaluate carefully.

Nghệ thuật là đặt mỗi task vào bậc phù hợp.

Ma Trận Lựa Chọn

Tôi dùng cây quyết định bốn câu hỏi cho mỗi agent hoặc task mới:

Câu hỏi 1: Có cần suy luận hoặc phán xét thực sự không?

“Suy luận” nghĩa là task đòi hỏi cân nhắc thông tin mơ hồ, đưa ra quyết định với dữ liệu không đầy đủ, hiểu context và sắc thái, hoặc giải quyết vấn đề không có giải pháp thuật toán rõ ràng.

Ví dụ cần suy luận: - Chẩn đoán tại sao một thành phần hạ tầng đang hoạt động bất thường - Quyết định liệu lỗ hổng bảo mật có cần hành động ngay hay có thể chờ - Lập kế hoạch triển khai tính năng phức tạp với nhiều đánh đổi - Viết code đòi hỏi hiểu logic nghiệp vụ, không chỉ cú pháp

Ví dụ KHÔNG cần suy luận: - Phân loại ticket hỗ trợ theo danh mục - Trích xuất dữ liệu có cấu trúc từ văn bản không định dạng - Dịch nội dung giữa các định dạng (markdown sang HTML, JSON sang CSV) - Tạo biến thể của nội dung hiện có

Nếu có → bắt đầu ở cấp Sonnet hoặc Opus. Nếu không → bắt đầu ở cấp Haiku hoặc Flash.

Câu hỏi 2: Blast radius của lỗi là gì?

Nếu agent đang thực hiện hành động với hậu quả thực tế (sửa đổi file, gọi API, gửi tin nhắn, deploy dịch vụ), lỗi tốn hơn chỉ chi phí token. Nó tốn thời gian khắc phục, có thể mất dữ liệu, có thể ảnh hưởng người dùng.

Task blast radius cao xứng đáng với khả năng model cao hơn. Không phải vì model thông minh hơn không mắc lỗi — chúng vẫn mắc — mà vì chúng mắc ít lỗi hơn trong những cuộc gọi suy luận và phán xét quan trọng.

Nếu blast radius cao → không thỏa hiệp về khả năng model. Opus cho hạ tầng production. Nếu blast radius thấp → tối ưu hóa chi phí tích cực là an toàn.

Câu hỏi 3: Khối lượng yêu cầu là bao nhiêu?

Khối lượng nhân lên chi phí. Model tốn gấp 3 lần mỗi yêu cầu sẽ tốn gấp 3 lần khi bạn chạy 10.000 yêu cầu mỗi ngày. Kinh tế thay đổi đáng kể ở quy mô lớn.

Nguyên tắc kinh nghiệm: - Dưới 1.000 yêu cầu/ngày: chất lượng model quan trọng hơn giá - 1.000-100.000 yêu cầu/ngày: cân bằng chất lượng và giá cần thận - Trên 100.000 yêu cầu/ngày: mỗi \$0,01/yêu cầu là \$1.000/ngày

Ở khối lượng cao, cách tiếp cận đúng gần như luôn là dùng model rẻ nhất đáp ứng ngưỡng chất lượng, đo chất lượng theo kinh nghiệm, và nâng cấp nếu chỉ số chất lượng giảm dưới ngưỡng.

Câu hỏi 4: Có cần context window dài không?

Nếu task đòi hỏi đưa lượng dữ liệu lớn cho model — toàn bộ codebase, tài liệu lớn, lịch sử hội thoại mở rộng — bạn cần model hỗ trợ nó mà không có hình phạt.

Cẩn thận với: hệ số nhân 2x của Gemini trên 200K, nhà cung cấp tính phụ phí cho context dài, và model có giới hạn context cứng buộc bạn cắt bớt.

Lựa chọn context dài tốt nhất không có hình phạt: GPT-4.1 (1M, không phụ phí), Llama 4 Maverick (1M, rẻ), Claude 4.x với prompt caching.

Câu Chuyện Chiến Trường: Định Tuyển Task Hạ Tầng Đến Sai Model

Điều này ngại ngùng để viết, nhưng nó minh họa rõ dạng lỗi.

Thiết Lập

Đầu năm 2025, khi tôi lần đầu dựng hạ tầng đa agent, tôi xây API proxy routing để mặc định dùng Gemini 2.0 Flash cho hầu hết task. Flash rẻ, nhanh và đủ khả năng cho phần lớn những gì tôi cần. Tôi có quy tắc định tuyển task phức tạp đến model có khả năng hơn, nhưng tôi chưa suy nghĩ đầy đủ về phân loại.

“Task quản lý hạ tầng” được phân loại là “phức tạp” và được định tuyển đến... Gemini 2.0 Pro. Không phải Claude Opus. Thậm chí không phải GPT-4.

Lý do là chi phí. Lúc đó, Gemini Pro rẻ hơn đáng kể so với Opus. Tôi tự thuyết phục rằng nó đủ tốt cho debug hạ tầng.

Điều Đã Xảy Ra

Trong ba tuần, tôi tích lũy một mẫu: các agent xử lý vấn đề hạ tầng cứ bị kẹt. Không phải thảm họa — chúng sẽ không phá vỡ mọi thứ — nhưng chúng sẽ đến điểm quyết định, đưa ra phản hồi mơ hồ, và trả vấn đề lại cho tôi kèm tóm tắt đúng nhưng vô ích.

Ví dụ cụ thể: Node-02 có timeout kết nối gián đoạn đến instance Redis. Agent được cấp quyền truy cập server log, cấu hình Redis và tool chẩn đoán mạng. Nó chạy qua playbook chẩn đoán tiêu chuẩn, xác định ba nguyên nhân có thể và kết luận: “Vấn đề có thể liên quan đến cấu hình mạng, cài đặt timeout Redis, hoặc connection pooling của ứng dụng. Khuyến nghị điều tra thêm.”

Đó là câu trả lời đúng. Nó cũng vô dụng. Kỹ sư hạ tầng giỏi — hoặc agent có khả năng — đáng lẽ phải xếp hạng các nguyên nhân đó theo khả năng dựa trên bằng chứng log, kiểm tra cái có khả năng nhất, và giải quyết hoặc giải thích chính xác tại sao không.

Tôi chuyển routing hạ tầng sang Claude Opus 4.6. Cùng log, cùng tool, cùng prompt. Agent đó: 1. Xác định rằng mẫu timeout tương quan với các sự kiện CPU cao trên application server (không phải Redis) 2. Nhận ra đây là trường hợp cổ điển của connection pool exhaustion dưới áp lực CPU 3. Đề xuất thay đổi pool size cụ thể kèm lý luận 4. Soạn thảo thay đổi cấu hình để tôi xem xét

Cùng task. Kết quả khác. Sự khác biệt không phải là trí thông minh trừu tượng — mà là khả năng của model tổng hợp bằng chứng mơ hồ thành kết luận cụ thể, có thể thực hiện.

Bài Học

Gemini Flash (và các model tối ưu chi phí tương tự) xuất sắc với task có câu trả lời đúng rõ ràng. Trích xuất, định dạng, phân loại, tạo từ template — những thứ này có tính đúng đắn khách quan mà model rẻ hơn xử lý tốt.

Debug hạ tầng không phải task có câu trả lời đúng rõ ràng. Nó đòi hỏi hình thành giả thuyết, cân nhắc bằng chứng, xem xét thông tin còn thiếu và đưa ra phán xét. Đây là nơi khoảng cách khả năng giữa Gemini Flash và Claude Opus rõ ràng nhất.

Sự chênh lệch chi phí là thực: Gemini Flash \$0,30/MTok input so với Opus \$5/MTok là chênh lệch 16 lần. Nhưng nếu Gemini Flash tiêu tốn của tôi hai giờ debug mà Opus đã giải quyết trong mười lăm phút, kinh tế học nghiêng mạnh về Opus.

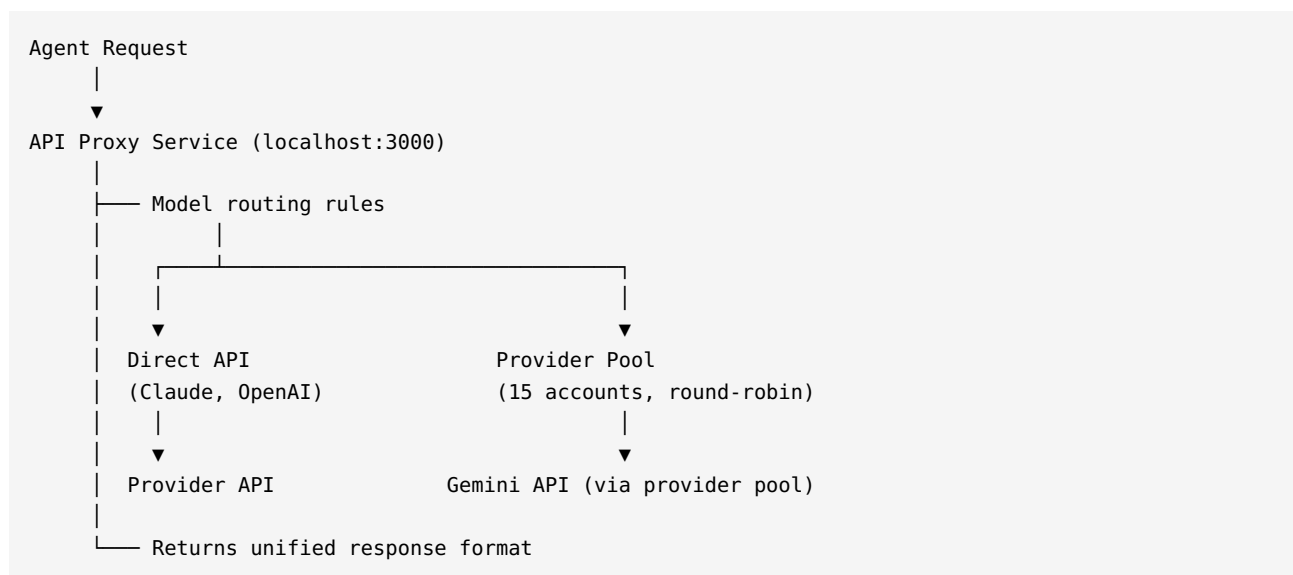
Luôn tính chi phí model so với tổng chi phí của task, bao gồm thời gian kỹ sư.

Câu Chuyện Chiến Trường: API Proxy Round-Robin Với 15 Tài Khoản Nhà Cung Cấp

Hãy để tôi mô tả hệ thống routing thực tế tôi chạy, vì nó minh họa một số nguyên tắc lựa chọn model trong thực tế.

Kiến Trúc

API proxy của chúng tôi là dịch vụ cục bộ chạy trên máy trạm của tôi mà tất cả agent đều dùng như endpoint AI. Nó cung cấp interface API thống nhất tương thích OpenAI và xử lý routing phía sau.



Quy tắc routing được cấu hình theo model prefix và task tag:

```
# API proxy 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 – provider pool for Gemini
  - match:
    model_prefix: "gemini"
    backend: provider_pool
    failover: direct_google # Fall back to direct Google if pool exhausted

  # Monitoring and health checks – cheapest available
  - match:
    tags: ["health_check", "monitoring"]
    backend: provider_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: provider_pool
  rate_limit:
    requests_per_minute: 20
    max_concurrent: 3

pool_config:
  provider:
    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 provider accounts exhausted"
      notify: telegram

```

Code quản lý pool:

```

class ProviderPool:
    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 provider accounts quota exhausted. "
            "Gemini-routed requests will fail until quota resets."
        )

```

Ngày Tất Cả 15 Tài Khoản Chạm 100% Đồng Thời

Tôi đã đề cập điều này ở Chương 1 nhưng muốn đi sâu hơn vào khía cạnh lựa chọn model ở đây.

Job batch đang chạy tạo metadata ảnh cho gần 300 bài blog. Task là: cho tiêu đề và tóm tắt bài blog, tạo ba query tìm kiếm ảnh, prompt tạo ảnh Gemini, alt text và metadata SEO. Output JSON có cấu trúc.

Đây chính xác là loại task Flash giỏi. Output có cấu trúc, định dạng rõ ràng, không cần suy luận mơ hồ. Lựa chọn model của tôi (Flash) là đúng.

Điều sai là ước tính tiêu thụ. Tôi đã ước tính ~400 token mỗi yêu cầu (input + output). Thực tế là ~1.200 token mỗi yêu cầu, do:

- System prompt dài hơn tôi nhớ (tôi đã mở rộng nó hai tuần trước)
- JSON output schema thêm vào mỗi yêu cầu (~180 token)
- Tạo JSON dài dòng của Gemini (nhiều khoảng trắng hơn, chuỗi dài hơn so với ước tính)
- Hệ số nhân 2x kích hoạt vì lịch sử hội thoại đẩy tôi qua 200K token giữa chừng batch

Tiêu thụ thực tế: $\sim 300 \text{ bài} \times 1.200 \text{ token} \times 4 \text{ lời gọi API mỗi bài} = \text{hơn } \mathbf{1.400.000 \text{ token}}$.

Tiêu thụ dự kiến (ước tính của tôi): $\sim 300 \times 400 \times 4 = \text{dưới } \mathbf{500.000 \text{ token}}$.

Gấp 3 lần tiêu thụ dự kiến trên 15 tài khoản đồng thời. Tôi không tính đến thời điểm reset quota hàng ngày — tất cả 15 tài khoản đã reset vào nửa đêm hôm trước, nghĩa là tất cả đều có cùng lượng quota trống và tất cả đạt trần vào thời điểm tương tự.

Giải pháp tôi triển khai sau đó:

```
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

```

Câu Chuyện Chiến Trường: Bug Không Khớp Model (HTTP 400 INVALID_ARGUMENT)

Cái này mất hai tiếng để debug và hoàn toàn do tôi tự gây ra.

Thiết Lập

Gateway agent được cấu hình dùng `claude-opus-4-5-thinking` — một model suy luận tôi đang test. File config đã được đặt thủ công và tôi đã quên về nó khi tôi chuyển lưu lượng API của gateway agent qua router proxy cục bộ.

Router proxy có quy tắc định tuyến: nếu tên model bắt đầu bằng `claude`, định tuyến trực tiếp đến Anthropic API. Hợp lý.

Nhưng tôi cũng đang thử nghiệm dùng provider pool cho một số lưu lượng Claude qua compatibility layer. Compatibility layer được cấu hình để dịch lời gọi Claude API sang lời gọi Gemini API, cho phép Gemini phục vụ các yêu cầu định dạng Claude (với mức giảm giá đáng kể).

Điều Đã Xảy Ra

Logic routing chạy như sau:

1. Gateway agent yêu cầu `claude-opus-4-5-thinking`
2. Router: bắt đầu bằng `claude` → Anthropic trực tiếp... nhưng khoan, flag tương thích provider pool đang bật
3. Router (với flag): dịch yêu cầu Claude sang định dạng Gemini → định tuyến đến provider pool
4. Translation layer: dịch tên model sang `gemini-2.5-pro` (tương đương gần nhất)
5. Nhưng: yêu cầu gốc bao gồm tham số `reasoning_effort: "high"` (đặc thù Anthropic cho model thinking)
6. Gemini API nhận yêu cầu với tham số `reasoning_effort`
7. Gemini API trả về HTTP 400 INVALID_ARGUMENT

Thông báo lỗi:

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

Con đường debug: 1. Gateway agent báo lỗi 400 lặp đi lặp lại, không thể hoàn thành task 2. Tôi kiểm tra log gateway agent — 400 INVALID_ARGUMENT, model: gemini-2.5-pro 3. Tôi kiểm tra config gateway agent — model được đặt là claude-opus-4-5-thinking 4. Tôi kiểm tra proxy log — translation đang xảy ra, reasoning_effort không bị lược bỏ 5. Tôi kiểm tra translation code — nó dịch tên model nhưng không lược bỏ tham số đặc thù nhà cung cấp

Cách sửa:

```
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
```

Bài Học Lớn Hơn

Tham số model đặc thù nhà cung cấp không dịch chuyển qua các nhà cung cấp. Nghe hiển nhiên nhìn lại. Không hiển nhiên khi tôi xây compatibility layer lúc 11 giờ đêm.

Nếu bạn xây bất kỳ abstraction layer nào định tuyến giữa các nhà cung cấp model, bạn phải: 1. Lược bỏ tham số đặc thù nhà cung cấp trước khi định tuyến đến nhà cung cấp khác 2. Ghi log khi bạn lược bỏ tham số (hành vi thay đổi im lặng nếu không) 3. Hiểu tham số nào đặc thù nhà cung cấp so với phổ quát 4. Test translation layer với các yêu cầu bao gồm tham số edge-case

Tham số đặc thù nhà cung cấp cần chú ý:

Nhà cung cấp	Tham số cụ thể	Ghi chú
Anthropic	reasoning_effort, thinking, betas, anthropic_version	Kiểm soát model thinking
OpenAI	response_format với strict mode, store, prediction	Không phải tất cả tương thích Gemini
Google	generation_config.response_mime_type, safety_settings	Safety đặc thù Google
OpenAI reasoning	reasoning_effort (o3/o4), reasoning_summary	Ý nghĩa khác với Anthropic

Xây Dựng Chính Sách Lựa Chọn Model

Đây là chính sách thực tế tôi chạy, được biểu diễn dưới dạng 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,  
}
```

Kiểm Tra Thực Tế Mã Nguồn Mở

DeepSeek, Llama và Mistral xứng đáng được đánh giá trung thực. Sự cường điệu xung quanh model mã nguồn mở đôi khi quảng cáo quá mức về mức độ sẵn sàng production của chúng.

Những gì model mã nguồn mở làm tốt: - Task khối lượng cao nhạy cảm chi phí nơi lỗi đôi khi chấp nhận được - Task có câu trả lời đúng rõ ràng (trích xuất, phân loại, định dạng) - Deployment riêng tư nơi dữ liệu không thể rời khỏi hạ tầng của bạn - Thử nghiệm và phát triển nơi độ tin cậy production không được yêu cầu

Điểm yếu cho workload agent: - Độ tin cậy tool use thấp hơn model frontier ở mức chất lượng tương đương - Tính nhất quán tuân theo hướng dẫn biến động hơn - Sự kết hợp context dài suy giảm nhanh hơn - Debug các quirk đặc thù model thêm chi phí kỹ thuật

DeepSeek R1 (MoE 671B ở \$0,55/\$2,19) thực sự cạnh tranh với model frontier cấp trung cho task suy luận. Tôi đã dùng nó cho task tạo code nơi tôi cần bảo mật (dữ liệu không rời hạ tầng) và chất lượng chấp nhận được.

DeepSeek V3.2 (\$0,14/\$0,28) là option chi phí cực đoan nhất tôi dùng trong production. Tôi dùng nó cho task nơi “khá tốt” là đủ và khối lượng rất cao. Nó thất bại nhiều hơn Gemini Flash về tool use, nên tôi không bao giờ dùng nó cho agent cần gọi tool đáng tin cậy.

Mistral Nemo (\$0,02/M) rẻ đến mức gần như miễn phí. Tôi dùng nó cho task phân loại nội bộ nơi tôi có thể chịu đựng lỗi và con người xem xét bắt lỗi. Tôi không dùng nó cho bất cứ thứ gì hướng đến khách hàng hay có tác động phụ.

Theo Dõi Lựa Chọn Model Theo Thời Gian

Lựa chọn model không phải quyết định một lần. Model cải thiện, giá thay đổi, workload của bạn phát triển. Xây đo lường vào hệ thống từ ngày đầu.

```
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()]
```

Chạy báo cáo này hàng tuần. Bạn sẽ tìm thấy bất ngờ: model bạn nghĩ đang hoạt động tốt có tỷ lệ thành công thấp hơn bạn nhận ra. Model bạn đánh giá thấp đang xử lý task tốt hơn các lựa chọn đắt tiền hơn. Thay đổi giá khiến lựa chọn hiện tại của bạn không còn tối ưu.

Tham Khảo Nhanh: Quy Tắc Lựa Chọn

Cho khoảnh khắc bạn cần câu trả lời nhanh:

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

Khi nghi ngờ, hãy chạy thử nghiệm nhỏ. Lấy 20 ví dụ đại diện từ workload thực tế của bạn. Chạy chúng qua model rẻ và model đắt. So sánh output. Sự chênh lệch chất lượng thường nhỏ hơn — hoặc lớn hơn — so với bạn mong đợi, và dữ liệu thực tế đánh bại suy đoán.

Điểm Mấu Chốt

- Lựa chọn model là bài toán tối ưu hóa chi phí-chất lượng, không phải câu hỏi “cái nào thông minh nhất”
- Ma trận lựa chọn: độ phức tạp suy luận, blast radius của lỗi, khối lượng yêu cầu, nhu cầu context window
- **Claude Opus 4.6 bắt buộc cho debug hạ tầng, phân tích bảo mật và agent đa bước phức tạp** — khoảng cách khả năng là có thực và chi phí thời gian của lỗi vượt quá chênh lệch chi phí token
- **Gemini 2.5 Flash là vô địch giá-hiệu suất** cho task khối lượng cao, câu trả lời đúng rõ ràng — nhưng chú ý hệ số nhân 2x trên 200K token
- **GPT-4.1** có kinh tế học context dài tốt nhất (1M token, không phụ phí) cho workload cần window lớn
- Kiến trúc routing + provider pool hoạt động nhưng cần ước tính quota chính xác và circuit breaker ở phía tiêu thụ
- Bug không khớp model (tham số đặc thù nhà cung cấp vượt qua ranh giới API) là dạng lỗi thực — xây dựng lược bỏ tham số vào bất kỳ translation layer nào
- Đo hiệu suất model theo kinh nghiệm với workload thực tế — bất ngờ theo cả hai hướng là phổ biến

Tiếp theo: Chương 4 — Thiết Kế Hệ Thống Agent Sống Sót Khi Chạm Với Thực Tế

PHẦN 2: KIẾN TRÚC

Chương 4: Các Mẫu Kiến Trúc Agent

“Kiến trúc đúng không phải là cái thông minh nhất. Đó là cái vẫn hoạt động lúc 3 giờ sáng khi có gì đó bốc cháy.”

Giới Thiệu

Khi tôi lần đầu deploy AI agent trong production, tôi đã mắc sai lầm mà mọi người đều mắc: tôi coi chúng như microservice. Cấp cho chúng một API, trở đến một số tool, xong. Ship nó.

Cách đó hoạt động ổn cho đến lần đầu tiên một agent cần phối hợp với agent khác. Sau đó nó cần theo dõi hạ tầng. Rồi nó cần phục hồi sau lỗi của chính mình. Sau đó tôi có ba agent theo dõi nhau trên ba server và không cái nào đồng ý về ai đang phụ trách.

Chương này nói về các mẫu kiến trúc tôi học được theo cách khó. Không phải mẫu lý thuyết từ các bài báo. Mà là những mẫu đã sống sót khi chạm với production.

Chúng ta sẽ đề cập năm mẫu chính, sắp xếp sơ bộ từ đơn giản đến phức tạp:

1. **Agent Đơn** — một agent, một công việc
2. **Điều Phối Đa Agent** — một coordinator với các sub-agent
3. **Agent Mesh** — các peer theo dõi và phục hồi nhau
4. **Agent Hướng Sự Kiện** — agent được đánh thức bởi sự kiện real-time
5. **Kiến Trúc Hybrid** — kết hợp các mẫu trong production

Cho mỗi mẫu, tôi sẽ cho bạn chi tiết triển khai thực tế, các dạng lỗi tôi gặp phải, và những gì tôi sẽ làm khác đi.

Mẫu 1: Agent Đơn (Claude Code Standalone)

Đây là nơi mọi người bắt đầu. Một agent, một phiên, một task.

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

Khi Nào Nó Hoạt Động

Agent đơn bị đánh giá thấp. Cho một số lượng task đáng ngạc nhiên, nó chính xác là đúng:

- Refactor codebase
- Viết và chạy script migration batch
- Debug sự cố production (với tool chỉ đọc)
- Tạo boilerplate hoặc tài liệu

Sự đơn giản là tính năng. Một tiến trình, một file log, một thứ để debug khi có gì đó sai.

Ràng Buộc Thực Sự: Context Window

Context window là toàn bộ bộ nhớ làm việc của agent đơn. Claude Opus 4.6 có 200K token. Nghe có vẻ nhiều cho đến khi bạn nhận ra:

```
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
```

Phiên bắt đầu ở 10K token đã dùng có thể đạt 150K sau một giờ làm việc nghiêm túc. Khi tiến gần đến giới hạn, hành vi của agent suy giảm. Nó bắt đầu quên context từ đầu hội thoại. Nó mắc lỗi mà nó đã không mắc lúc ban đầu.

Tôi đã có các phiên Claude Code đạt 97% quota và bị kill bằng signal 9. Không shutdown graceful, không checkpoint. Công việc chỉ dừng lại.

Nguyên tắc kinh nghiệm: Lập kế hoạch cho phiên agent đơn xử lý không nhiều hơn những gì vừa thoải mái trong 100K token hội thoại. Ngoài đó, bạn cần kiến trúc khác.

Anti-Pattern Agent Đơn

Lỗi super-prompt. Ban đầu tôi cố gắng viết một prompt khổng lồ xử lý toàn bộ deployment phức tạp: cập nhật database schema, migrate dữ liệu, cập nhật code ứng dụng, khởi động lại dịch vụ, chạy smoke test. Agent bắt đầu tốt, rồi ảo giác giữa chừng vì có quá nhiều thứ để theo dõi.

Cách sửa là chia thành bốn prompt tuần tự với handoff rõ ràng: 1. Phase 1: Analyze current state and plan migration 2. Phase 2: Execute database changes (sau khi xem xét output phase 1) 3. Phase 3: Update application code 4. Phase 4: Deploy and validate

Mỗi phase bắt đầu mới với context có mục tiêu. Agent biết chính xác những gì nó cần biết.

Bẫy đường dẫn tương đối. Agent đơn chạy trong các phiên sẽ chết. Nếu bạn tham chiếu `./configs/production.yaml`, nó hoạt động tốt trong phiên hiện tại. Nhưng nếu phiên agent bị kill và khởi động lại, hoặc nếu bạn spawn một sub-agent, thư mục làm việc có thể khác.

Luôn dùng đường dẫn tuyệt đối trong hướng dẫn agent. Luôn luôn. Tôi đã mất nhiều giờ debug lỗi chỉ là `./config` không phân giải từ thư mục làm việc khác.

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

# Good - works everywhere
/opt/agents/project/scripts/deploy.sh
```

Mẫu 2: Điều Phối Đa Agent

Khi agent đơn chạm giới hạn, bước tự nhiên tiếp theo là mô hình orchestrator-subagent. Một agent điều phối; nhiều agent chuyên biệt thực hiện công việc.

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

Cách Nó Hoạt Động Trong Thực Tế

Orchestrator nhận task phức tạp và chia thành các workstream song song hoặc tuần tự. Mỗi sub-agent nhận context tập trung: chỉ những gì nó cần để hoàn thành phần cụ thể của mình.

Đây là cách tôi có năm agent viết năm phần của cuốn sách đồng thời. Orchestrator xác định các chương, giao một chương cho mỗi agent với cùng style guide, chạy song song và gộp output. Hai mươi lăm phút cho bản thảo đầy đủ. Làm tuần tự trong một phiên sẽ mất nhiều giờ và có thể chạm giới hạn context.

Ngân Sách Context Sub-Agent

Mỗi sub-agent có context 200K token mới của riêng nó. Đây là lợi thế chính so với agent đơn cho task lớn: bạn không chiến đấu với context co lại qua một phiên dài. Mỗi agent bắt đầu sạch sẽ.

Ràng buộc là thiết kế task. Bạn phải cấu trúc task của mỗi sub-agent để nó có thể hoàn thành trong: - Context window riêng (200K) - Không cần phối hợp real-time với agent khác (trừ khi bạn thiết kế rõ ràng cho điều đó)

Quy tắc thực tế: Thiết kế task sub-agent để độc lập. Nếu agent B cần output từ agent A, đừng chạy song song — chuỗi tuần tự, với output của A được truyền rõ ràng trong prompt của B.

Mẫu Điều Phối Tôi Dùng

```
# 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: /opt/agents/project/
- Output: /opt/agents/project/output/agent-N/

Phần constraints là quan trọng. Sub-agent có năng lực và nhiệt tình. Không có ranh giới rõ ràng, chúng sẽ “giúp đỡ” bằng cách làm những thứ bạn không yêu cầu. Tôi đã có agent tái cấu trúc thư mục, cập nhật file config, cài đặt dependency — tất cả trong khi hoàn thành task được giao thành công và phá vỡ hoàn toàn môi trường cho agent tiếp theo.

Câu Chuyện Chiến Trường: Tác Nghẽn 15 Agent

Tôi từng spawn 15 sub-agent song song để xử lý 15 chương đồng thời. Mỗi agent được cho là viết lại một chương. Điều tôi không tính đến: ba trong số chúng cần cùng một file config chia sẻ. Tất cả ba đều ghi vào nó đồng thời. File kết quả bị hỏng. Cả ba chương đều thất bại, và tôi mất bốn mươi phút tháo gỡ conflict.

Bài học: Xác định tài nguyên chia sẻ trước khi spawn agent song song. Hoặc cấp cho mỗi agent bản sao riêng, hoặc serialize các agent chạm tài nguyên chia sẻ.

Vấn Đề Dọn Dẹp

Các phiên sub-agent không miễn phí. Chúng tiêu thụ quota. Nếu phiên agent bị treo hoặc bị orphan — có thể nó bị kẹt trên lời gọi mạng, có thể nó mất kết nối — nó giữ phiên tmux sống và tiếp tục đốt quota API.

Tôi phát hiện ra điều này theo cách khó khi tìm thấy bảy phiên tmux orphan đang chạy lặng lẽ. Không cái nào làm gì hữu ích. Chúng chỉ sống, và heartbeat prompt của chúng tiếp tục kích hoạt.

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

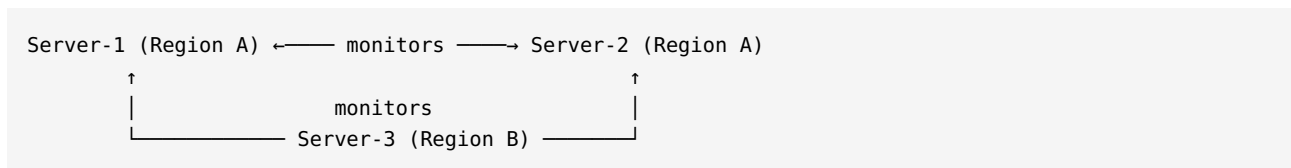
# Kill them
tmux kill-session -t agent-3
```

Sau sự cố này tôi thêm chính sách cleanup: bất kỳ sub-agent nào được spawn theo chương trình nhận flag `cleanup: "delete"` và timeout wall-clock tối đa. Nếu nó không hoàn thành trong timeout, nó bị kill.

Mẫu 3: Agent Mesh

Đây là nơi thú vị — và nơi tôi dành ba tuần lặp lại trên một thiết kế thực sự hoạt động.

Agent mesh là một nhóm peer agent theo dõi nhau và có thể phục hồi lỗi của nhau. Không có agent đơn nào đặc biệt. Bất kỳ agent nào có thể quan sát tình trạng của bất kỳ agent nào khác. Bất kỳ agent nào có thể cố gắng sửa bất kỳ agent nào khác.



Trong thiết lập production của chúng tôi, chúng tôi chạy nhiều server trên các vùng địa lý. Mỗi server chạy gateway agent. Mỗi server chạy cùng script mesh monitor trên heartbeat 3 phút. Mỗi agent có thể khởi động lại bất kỳ agent nào khác qua SSH.

Tiến Hóa: v1 → v2.3

Để làm đúng cần ba lần sửa đổi chính. Hãy để tôi đi qua những gì đã hỏng và tại sao.

v1: Không Phối Hợp

Phiên bản đầu đơn giản: nếu agent A thấy agent B bị ngã, agent A khởi động lại B. Nếu agent C cũng thấy B bị ngã cùng lúc, agent C cũng cố khởi động lại B. Cả hai thành công, hoặc chúng conflict. Dù thế nào, bạn nhận được các lần thử khởi động lại trùng lặp và log nhiều.

Ổn cho lỗi thỉnh thoảng. Tệ hại khi sự cố mạng khiến tất cả agent thấy nhau bị ngã đồng thời và chúng bắt đầu khởi động lại mọi thứ.

v2.2: Bầu Chọn Leader theo IP

Thêm bầu chọn leader: agent với địa chỉ IP thấp nhất là “leader” và là người duy nhất được phép thực thi khôi phục.

Cách này hoạt động cho đến khi tôi nhận ra Server-2 không bao giờ có thể là leader, vì hai server khác đều có IP thấp hơn. Server-2 sẽ phát hiện lỗi và chẩn đoán đúng, nhưng không bao giờ kích hoạt khôi phục. Nó là hành khách trên mesh của chính nó.

Câu hỏi phơi bày bug này: “Vai trò của Server-2 trong mesh là gì?” Câu trả lời — “nó quan sát nhưng không bao giờ hành động” — làm vấn đề hiển nhiên.

v2.3: Pure Flock

Giải pháp là bỏ hoàn toàn bầu chọn leader và dùng kernel-level file locking (flock) trực tiếp.

Cách phối hợp dựa trên flock hoạt động:

1. Mỗi agent, khi nó phát hiện lỗi, cố gắng lấy exclusive lock trên file lock khôi phục của máy mục tiêu.
2. Lock được lấy qua SSH: máy remote giữ file descriptor.
3. Vì lock được giữ bởi tiến trình SSH, nó tự động được giải phóng khi phiên SSH kết thúc — dù đó là hoàn thành bình thường, crash hay mất mạng.
4. Kernel đảm bảo chỉ một tiến trình giữ lock tại một thời điểm, không có race condition.

```
# 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
```

Vẻ đẹp của cách tiếp cận này:

- **Không có orphan lock.** SSH chết → tiến trình bash chết → file descriptor đóng → kernel giải phóng lock. Không cần cleanup.
- **Không có phức tạp bầu chọn leader.** Bất kỳ agent nào có thể sửa bất kỳ agent nào. Cái đầu tiên lấy lock thắng.

- **Atomic ở cấp kernel.** Không có khoảng trống giữa “kiểm tra xem đã lock chưa” và “lấy lock.”
- **Backpressure tự nhiên.** Nếu việc sửa mất 5 phút, không agent nào khác lãng phí tài nguyên thử cùng việc sửa trong 5 phút đó.

Script Mesh Monitor Đầy Đủ

Đây là script production (v2.3, đơn giản hóa để rõ ràng):

```
#!/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="${AGENT_NAME:-unknown}"
FAIL_THRESHOLD=2
HEARTBEAT_DIR="/opt/agents/workspace"
TELEGRAM_TOKEN="${TELEGRAM_BOT_TOKEN}"
TELEGRAM_CHAT="${TELEGRAM_CHAT_ID}"
LOG_FILE="/tmp/mesh-monitor-${AGENT_NAME}.log"

# Peer configuration
declare -A PEERS=(
    ["agent-a"]="10.10.0.2"
    ["agent-b"]="10.10.0.3"
    ["agent-c"]="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}.txt"

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

```

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=$(ssh -o ConnectTimeout=10 -o StrictHostKeyChecking=no \
        "agent-user@${peer_ip}" bash << 'EOF' 2>/dev/null || echo "SSH_FAILED")

    issues=()

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

    # Check API proxy service
    if ! systemctl is-active api-proxy > /dev/null 2>&1; then
        issues+=("api-proxy: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=$(ssh -o ConnectTimeout=15 -o StrictHostKeyChecking=no \
        "agent-user@${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 gateway-agent
        sleep 5
        if systemctl --user is-active gateway-agent > /dev/null 2>&1; then
            fixed+=("gateway:restarted")
        fi
    fi

    if [[ "$issues" == *"api-proxy:inactive"* ]]; then
        systemctl start api-proxy
        sleep 5
        if systemctl is-active api-proxy > /dev/null 2>&1; then
            fixed+=("api-proxy: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 $FAIL_THRESHOLD times - triggering recovery"
        send_telegram "CRITICAL [$AGENT_NAME]: $peer_name is DOWN (fail $count/
$FAIL_THRESHOLD)"

        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 [[ "$recovery_result" == *"FIXED"* ]]; then
                send_telegram "RESOLVED [$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 agent
handles it

            else
                send_telegram "FAILED [$AGENT_NAME]: Could not fix $peer_name -
$recovery_result"
            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 Thay Vì Cron

Sau cuộc khủng hoảng tối ưu token (thêm về điều đó trong Chương 5), tôi chuyển từ chạy mesh monitor qua gateway agent sang chạy nó như pure systemd timer. Điều này có nghĩa là tiêu thụ zero AI token mỗi chu kỳ kiểm tra.

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

[Service]
Type=simple
User=agent-user
Environment="AGENT_NAME=agent-a"
Environment="TELEGRAM_BOT_TOKEN=xxx"
Environment="TELEGRAM_CHAT_ID=your_chat_id"
ExecStart=/opt/agents/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
```

Những Gì Mesh Không Thể Sửa

Hãy thành thật về giới hạn. Mesh monitor tôi mô tả có thể khởi động lại dịch vụ systemd và container Docker. Nó không thể:

- Sửa dịch vụ cấu hình sai mà crash ngay khi khởi động lại
- Phục hồi từ đầy ổ đĩa (nó có thể cảnh báo, nhưng dọn ổ đĩa đòi hỏi phân xét)
- Xử lý network partition nơi agent không đồng ý về ai đang ngã
- Sửa bất cứ thứ gì đòi hỏi input tương tác hoặc phân xét con người nhiều bước

Mesh giỏi trong “dịch vụ ngã xuống, khởi động lại nó.” Nó không phải sự thay thế cho người on-call.

Mẫu 4: Agent Hướng Sự Kiện

Thay vì chạy theo lịch trình, agent hướng sự kiện thức dậy để phản hồi các sự kiện real-time. Chi phí polling bằng không. Thời gian phản hồi dưới giây.

Trường hợp sử dụng kinh điển trong hạ tầng của tôi: sự kiện Docker.

```
# 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
```

Kiến Trúc Cảnh Báo Hai Lớp

Tôi chạy hai hệ thống cảnh báo độc lập. Chúng bổ sung cho nhau:

Lớp 1: Docker Events Listener (real-time, < 5 giây)

```
#!/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
```

```
    fi
done
```

Lớp 2: Prometheus + Alertmanager (dự phòng, độ trễ ~3 phút)

Prometheus scrape metric mỗi 15 giây và đánh giá quy tắc cảnh báo. Khi quy tắc kích hoạt, Alertmanager gửi cảnh báo đến webhook receiver, kích hoạt phiên gateway agent.

```
# 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"

# 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
    ])

```

Tại Sao Hai Lớp?

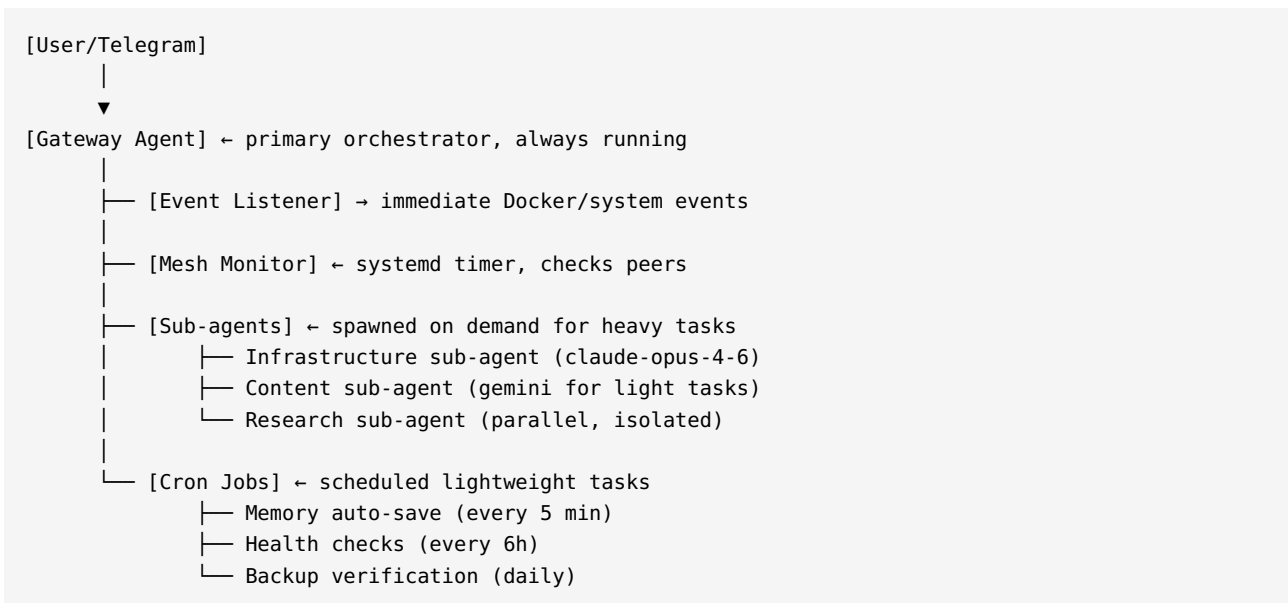
Vì các dạng lỗi không chồng chéo:

Lỗi	Lớp 1	Lớp 2
Container crash	Ngay lập tức	Sau 3 phút
Docker daemon treo	Im lặng	Prometheus phát hiện
Scrape metric thất bại	Vẫn cảnh báo	Không có dữ liệu
Network partition	Vẫn cục bộ	Có thể thất bại

Không hệ thống nào đủ tin cậy khi đứng một mình. Cùng nhau, cái này bao phủ điểm mù của cái kia.

Mẫu 5: Kiến Trúc Hybrid

Trong thực tế, bạn dùng kết hợp. Hệ thống production của tôi trông như thế này:



Lựa Chọn Model Theo Task

Không phải tất cả agent đều cần cùng một model. Tôi khớp model theo loại task:

```

# gateway-agent.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: "api-proxy/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"
```

Quy tắc cứng: **không bao giờ dùng model free-tier hoặc phụ thuộc quota cho khôi phục hạ tầng**. Sau khi chúng kiến provider pool quota chạm 100% trên tất cả 15 tài khoản đồng thời, tôi giữ các task hạ tầng quan trọng trên Anthropic direct API trả phí.

Hướng Dẫn Quyết Định Kiến Trúc

Đây là cách tôi sẽ chọn kiến trúc ngày nay:

```
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
```

Thuế Phối Hợp

Mỗi mẫu ngoài agent đơn đều thêm chi phí điều phối. Trước khi chọn kiến trúc phức tạp, hãy hỏi:

- Task có thực sự cần tính song song, hay tôi chỉ hứng khởi với distributed system?
- Tôi có thể chia khác đi để sub-agent thực sự độc lập không?
- Điều gì xảy ra khi một thành phần lỗi — hệ thống có suy giảm graceful không?

Tôi đã thấy kỹ sư (kể cả bản thân) dùng kiến trúc mesh đa agent khi một phiên agent đơn được prompt tốt đã có thể làm công việc trong một phần mười thời gian. Độ phức tạp là chi phí. Trả nó khi bạn phải, không phải như mặc định.

Danh Mục Dạng Lỗi

Lập tài liệu các dạng lỗi của bạn trước khi deployment, không phải sau. Mỗi mẫu kiến trúc có các dạng lỗi đặc trưng. Biết chúng trước nghĩa là phản hồi sự cố nhanh hơn.

Dạng lỗi Agent Đơn: - Chạm giới hạn context giữa chừng task (phiên suy giảm hoặc chết) - Phiên bị kill bởi signal 9 ở 97% quota - Đường dẫn tương đối bị hỏng sau khi thay đổi môi trường - Lỗi quyền tool giữa chuỗi với trạng thái một phần đã ghi

Dạng lỗi Điều Phối Đa Agent: - Sub-agent chạm giới hạn context 200K và dừng giữa task - Agent song song ghi vào cùng file đồng thời (hỏng dữ liệu) - Phiên sub-agent orphan tiếp tục đốt quota - Orchestrator mất dấu trạng thái sub-agent sau sự cố mạng - Sub-agent “giúp đỡ” ngoài phạm vi được giao (xem guardrail Chương 6)

Dạng lỗi Agent Mesh: - Network partition khiến tất cả agent thấy nhau bị ngã đồng thời (recovery storm) - File lock còn trên ổ đĩa nếu server hard-crash trước khi phiên SSH kết thúc (giảm thiểu bởi flock fd, không phải file) - Agent chẩn đoán đúng lỗi nhưng thiếu quyền để sửa - Agent healer mất kết nối giữa chừng khôi phục (mục tiêu được sửa một nửa) - Rotation SSH key trên một server phá vỡ khả năng SSH của tất cả server khác vào nó

Dạng lỗi Hướng Sự Kiện: - Docker events buffer overflow bị mất sự kiện dưới tải cực cao - Webhook receiver crash im lặng — không có cảnh báo nào kích hoạt cho đến khi ai đó nhận ra - Cảnh báo kích hoạt cho lần khởi động lại bảo trì có chủ ý (alert fatigue) - Hai sự kiện kích hoạt cho cùng container, hai lần thử khôi phục chạy song song

Biện pháp giảm thiểu:

```
# 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 "agent-user@$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 Trên Các Kiến Trúc

Mỗi mẫu cần observability khác nhau. Đây là những gì tôi instrument trong từng trường hợp:

Kiến trúc	Cần theo dõi	Tool
Agent Đơn	Sử dụng token phiên, thời gian wall-clock	Custom logging
Đa Agent	Tỷ lệ hoàn thành sub-agent, số phiên orphan	Session tracker
Agent Mesh	Độ trễ lấy flock, tỷ lệ thành công khôi phục	Prometheus metrics
Hướng Sự Kiện	Độ trễ xử lý sự kiện, sự kiện bị mất, false positive	Alertmanager

Cụ thể cho mesh, tôi phát metric từ script monitor:

```

# 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\""
```

Điều này cho bạn Grafana dashboard hiển thị agent nào đang chữa lành peer nào, tần suất lock contend, và nơi các lần thử khôi phục đang thất bại.

Điểm Mấu Chốt

1. **Agent đơn thường là đủ.** Khớp kiến trúc với yêu cầu thực tế.
 2. **200K token mỗi sub-agent là ngân sách của bạn.** Thiết kế task để vừa; đừng giả định agent tự quản lý context.
 3. **Dùng đường dẫn tuyệt đối.** Phiên chết. Phiên mới không biết thư mục làm việc của bạn.
 4. **Flock đánh bại bầu chọn leader** cho điều phối phân tán. Thời gian sống phiên SSH = thời gian sống lock = không có orphan lock.
 5. **Chuyển health check ra khỏi AI.** Pure bash systemd timer cho monitoring; AI chỉ cho chẩn đoán và khôi phục.
 6. **Hai lớp cảnh báo.** Docker events cho tốc độ; Prometheus cho đầy đủ.
 7. **Khớp model với task.** API trực tiếp trả phí cho hệ thống quan trọng. Model free/quota cho nội dung. Không bao giờ nhầm lẫn hai loại này.
 8. **Dọn dẹp sau bản thân.** Phiên agent orphan đốt quota im lặng. Triển khai timeout và chính sách cleanup.
-

Chương tiếp theo: Context Engineering — vì prompt bạn không tối ưu chính là token bạn đang trả tiền cho nó.

Chương 5: Context Engineering — Nghệ Thuật Tối Ưu Token

“Tôi cắt giảm 91% các file context của AI qua một đêm. Nó trở nên thông minh hơn.”

Giới Thiệu

Có một cuộc trò chuyện mà các kỹ sư DevOps ít khi nói đến: câu chuyện về những gì thực sự đi vào context window trước khi một dòng prompt nào của bạn được xử lý.

Hầu hết mọi người nghĩ về prompt engineering. Ít ai nghĩ đến context engineering — ngành kỹ luật về thiết kế, đo lường, và tối ưu hóa mọi thứ lấp đầy ngân sách token trước khi cuộc trò chuyện thậm chí bắt đầu.

Chương này nói về kỹ luật đó. Nó cũng kể về cách tôi học được điều này theo cách khó nhất: hơn một chục tài khoản provider bị tạm đình chỉ trong một ngày, ngân sách API hàng ngày về \$0, và một hệ thống AI production phải tiếp tục hoạt động.

Tôi sẽ chỉ cho bạn: - Những gì thực sự đang tiêu tốn ngân sách token của bạn - Câu chuyện giảm 91%: từ 9.030 → 797 token trong các file context - Cách thiết kế heartbeat làm hao hụt quota - Vấn đề orphan session mà không ai cảnh báo bạn - Cách sử dụng `CLAUDE.md` và `AGENTS.md` như các công cụ context engineering - Tại sao absolute path quan trọng hơn bạn nghĩ - Code để đo lường và quản lý ngân sách token của bạn

Context Engineering Là Gì?

Prompt engineering là viết tin nhắn bạn gửi đến model. Context engineering là thiết kế mọi thứ bao quanh tin nhắn đó — system prompt, tool definition, tài liệu được inject, lịch sử trò chuyện — sao cho mỗi token đều xứng đáng với vị trí của nó.

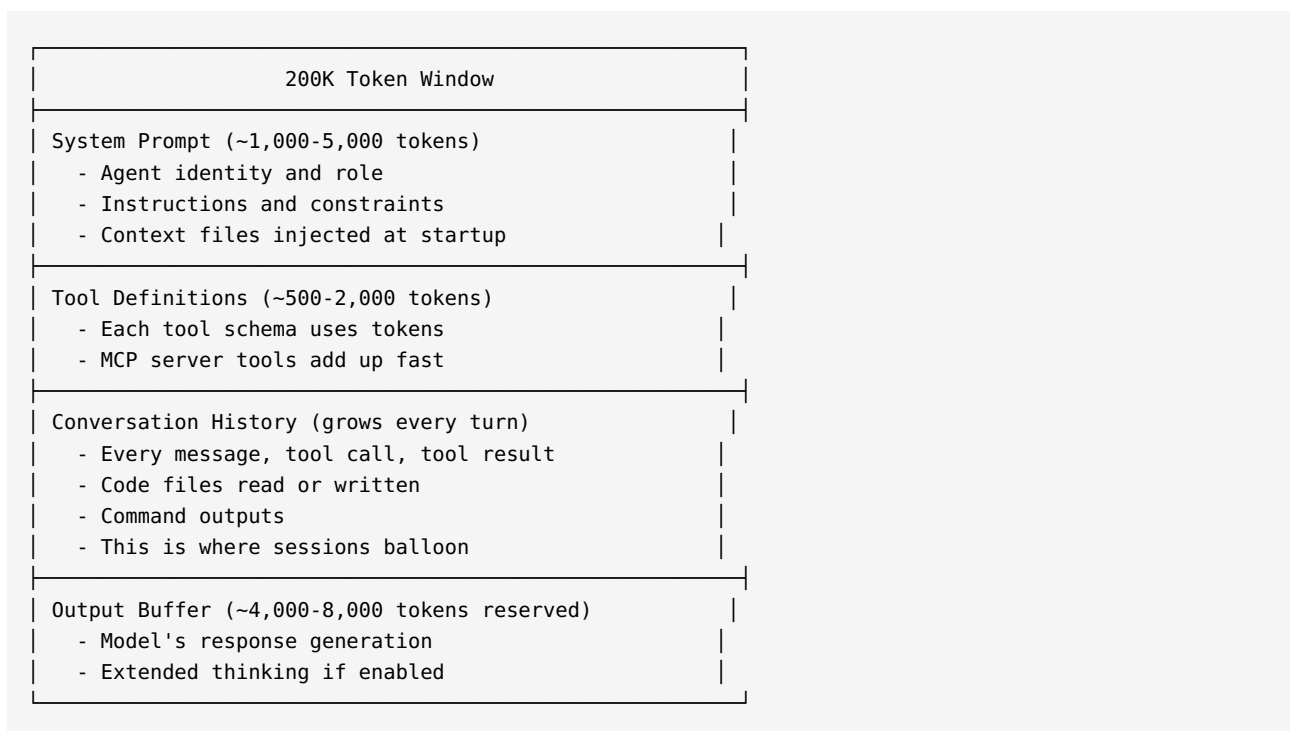
Hãy nghĩ như thế này: context window là một bàn làm việc có kích thước cố định. Bạn kiểm soát những gì được đặt lên bàn đó trước khi công việc bắt đầu. Nếu bạn chất đống sách tham khảo mà bạn không cần cho nhiệm vụ này, bạn đã lãng phí không gian có thể chứa thông tin thực sự liên quan đến nhiệm vụ.

Với context window 200K token: - 200K token nghe có vẻ vô hạn - 9.000 token trong các file context bị phình to nghe có vẻ nhỏ nhất - Cho đến khi bạn chạy 30 phiên agent mỗi ngày - Với \$15 mỗi triệu input token cho Opus 4.6 - Đó là \$0,135 mỗi phiên bị lãng phí trước khi có một từ hữu ích nào

Nhân với 30 phiên × 30 ngày = \$121,50/tháng bị đốt cháy vì context bloat. Cho một agent. Điều này quan trọng.

Cấu Trúc Tiêu Thụ Token

Trước khi tối ưu, bạn cần hiểu mình đang tối ưu cái gì. Đây là cách token thực sự bị tiêu thụ trong một phiên Claude Code / orchestrating agent điển hình:



Lịch sử trò chuyện là cỗ máy tăng trưởng tự nhiên. Mỗi lần gọi tool thêm input + output vào lịch sử. Đọc một file 500 dòng? Đó là ~2.000 token trong lịch sử. Chạy một lệnh xuất ra 200 dòng? Thêm ~800 token nữa. Một phiên code kéo dài một tiếng có thể dễ dàng tiêu thụ 100K token chỉ trong lịch sử trò chuyện.

Tuy nhiên, system prompt là biến số bạn có thể kiểm soát. Nó cố định ở mỗi phiên. Mỗi token bạn cắt từ system prompt là một token bạn tiết kiệm được ở mỗi phiên duy nhất.

Khủng Hoảng: Tài Khoản Provider Bị Tạm Đình Chỉ

Mọi chuyện bắt đầu với một cảnh báo lúc 2 giờ sáng ngày 11 tháng 2 năm 2026.

Tôi đang chạy orchestrating agent trên nhiều server, sử dụng hơn một chục tài khoản provider thông qua dịch vụ API proxy cục bộ của chúng tôi. Các tài khoản này cho phép truy cập vào các model Gemini qua pool token nội bộ của provider — thực chất là quota miễn phí trong giới hạn Google AI Pro.

Rồi Google tạm đình chỉ tất cả các tài khoản cùng một lúc.

Tôi đi từ chỗ có dung lượng AI không giới hạn sang chạy trên các tài khoản Anthropic native với giá theo token. Kinh tế thay đổi qua một đêm. Những thứ trước đây “miễn phí” bây giờ là tiền thật.

Tôi có hai lựa chọn: tắt các agent hoặc làm cho chúng hiệu quả hơn đáng kể. Tôi chọn hiệu quả.

Việc kiểm toán bắt đầu bằng câu hỏi: **những gì thực sự nằm trong context và kích hoạt mỗi phiên duy nhất?**

Kiểm Toán File Context

Orchestrating agent inject nhiều file vào system prompt khi khởi động. Tôi chưa bao giờ đo lường chúng. Đây là những gì tôi tìm thấy:

File	Token Cũ	Token Mới	Giảm
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%
TỔNG CỘNG	9.030	797	-91%

Chín nghìn token. Được inject trong mỗi phiên. Hầu hết là lặp lại từ các phiên bản trước, tích lũy chú thích, trường hợp ngoại lệ, và giải thích đã được thêm vào qua nhiều tháng mà không ai xóa đi.

Mức giảm 91% đến từ một nguyên tắc đơn giản: **context là lớp gợi ý, không phải lớp tài liệu.**

Kiến Trúc Context Ba Tầng

Sau khi kiểm toán, tôi tái cơ cấu các file context thành ba tầng dựa trên tầm suất thực sự cần thiết:

Tầng 1: Luôn Inject (System Prompt)

Những file này nhỏ, có tín hiệu cao, và thực sự cần thiết trong mỗi phiên. Chúng định nghĩa agent là ai và cách tìm những gì nó cần.

Mục tiêu: < 100 token mỗi file, tổng cộng < 800 token

```
<!-- MEMORY.md - SAU tối ưu: 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.
```

So sánh với bản gốc:

```
<!-- MEMORY.md - TRƯỚC tối ưu: 2,833 tokens -->
# Memory System

## Neural Memory
The gateway agent 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/agent-user/.agent-gateway/workspace/memory/`
These logs capture important decisions, configurations, and events.
```

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

Phiên bản bị phình to đang cố làm hướng dẫn sử dụng. Phiên bản tối ưu là con trỏ — nó nói với agent *nơi* tìm thông tin, không phải thông tin đó.

Tầng 2: Tham Chiếu Theo Yêu Cầu (Không Inject)

Đây là các tài liệu lớn mà agent chỉ tải khi có liên quan:

- docs/infrastructure.md — chi tiết server đầy đủ, địa chỉ IP, tham chiếu mật khẩu
- docs/architecture.md — phân tích sâu về thiết kế hệ thống
- Tài liệu kỹ năng — được tải khi một kỹ năng được gọi

Agent biết những thứ này tồn tại. Nó biết nơi để tìm chúng. Nó đọc chúng khi cần, không phải trước đó.

Tầng 3: Đã Lưu Trữ / Đã Xóa

Bất cứ điều gì nằm trong system prompt vì lý do lịch sử nhưng không còn cần thiết. Xóa nó đi. Sự cám dỗ giữ “phòng khi cần” là kẻ thù của hiệu quả context.

CLAUDE.md và AGENTS.md Như Công Cụ Context Engineering

Claude Code tự động inject các file CLAUDE.md từ thư mục gốc dự án và các thư mục cha vào system prompt của nó. Gateway agent có phiên bản tương đương của riêng mình: AGENTS.md.

Những file này là giao diện context engineering chính của bạn.

CLAUDE.md Nên Chứa Gì

```
# CLAUDE.md

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

## Quick Reference
- Repo root: /opt/agents/project/
- Deploy script: /opt/agents/project/scripts/deploy.sh
- Config: /opt/agents/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)
```

Chú ý những gì không có trong đó: tài liệu về cách thức hoạt động của mọi thứ, các quyết định lịch sử, giải thích về codebase. Những thứ đó thuộc về tài liệu mà agent có thể đọc khi cần — không phải trong mỗi context window.

Tỷ Lệ Instruction-to-Pointer

Một mô hình tư duy hữu ích: đo tỷ lệ instruction (phải làm gì) so với pointer (nơi tìm thông tin) trong các file context của bạn.

Context kém chủ yếu là instruction — nó cố gắng trả lời trước mọi câu hỏi có thể xảy ra. Context tốt chủ yếu là pointer — nó nói với agent nơi tìm câu trả lời khi cần.

Tỷ lệ mục tiêu: 30% instruction, 70% pointer

```
# Tệ: Nặng về instruction (nhiều token, đòn bẩy thấp)
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...

# Tốt: Nặng về pointer (ít token, đòn bẩy cao)
Database: see docs/infrastructure.md > Database section
Connection details and credentials in Vault: secret/db/production
Query patterns: docs/database-patterns.md
```

Tối Ưu Heartbeat: Từ 3 Phút → 30 Phút

Heartbeat là một tác vụ agent định kỳ kích hoạt theo lịch để duy trì trạng thái hệ thống, cập nhật bộ nhớ, kiểm tra sức khỏe hệ thống, và ghi file trạng thái.

Trước khi xảy ra khủng hoảng, gateway agent heartbeat chạy mỗi 3 phút.

Mỗi heartbeat kích hoạt một phiên agent đầy đủ. Mỗi phiên tiêu thụ system prompt đầy đủ (9.030 token trước khi tối ưu). Cộng thêm tool. Cộng thêm những gì heartbeat thực sự làm.

Hãy tính toán: - Khoảng cách 3 phút = 20 heartbeat/giờ - 24 giờ = 480 heartbeat/ngày - System prompt 9.030 token = 4,3M input token/ngày chỉ trong system prompt - Với \$15/M token = \$64,50/ngày chỉ trong heartbeat overhead

Đó là trước khi heartbeat làm bất cứ điều gì hữu ích.

Heartbeat Thực Sự Cần Gì

Câu hỏi đầu tiên là: heartbeat thực sự đang làm gì?

```
Heartbeat v1 (mỗi 3 phút):
- [x] Update HEARTBEAT.md với thời gian hiện tại
- [x] Kiểm tra xem có cron job nào thất bại không
- [x] Chạy neural-memory save cho các sự kiện gần đây
- [x] Kiểm tra dung lượng đĩa
- [x] Kiểm tra sức khỏe service
- [x] Lưu file bộ nhớ hàng ngày
```

Một số trong số này thực sự cần lý luận AI (chẩn đoán pattern lỗi). Hầu hết thì không (kiểm tra dung lượng đĩa, cập nhật timestamp).

Giải pháp là tách heartbeat:

Systemd timer (không AI, không token):

```
# 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 api-proxy
check_service agent-gateway
check_disk
update_heartbeat_file # Just write a timestamp
```

AI heartbeat (khoảng cách 30 phút, chỉ khi cần lý luận):

Heartbeat v2 (mỗi 30 phút):

- [x] Chạy neural-memory consolidation (AI reasoning)
- [x] Đánh giá pattern trong system health log (AI judgment)
- [x] Lưu ghi chú phiên có ý nghĩa (AI summarization)

Kết quả: 480 AI heartbeat/ngày → 48 AI heartbeat/ngày. Giảm 90% lượng token heartbeat tiêu thụ, không mất đi bất kỳ phạm vi giám sát nào.

Vấn Đề Orphan Session

Cái này đã khiến tôi tốn tiền thật trước khi tôi hiểu ra.

Khi một phiên AI agent bắt đầu rồi mất kết nối — network dropout, SSH disconnect, process bị OOM kill — phiên đó không phải lúc nào cũng tắt sạch. Trong các triển khai dựa trên tmux, phiên có thể tiếp tục chạy. Trong các triển khai dựa trên cron, lần gọi tiếp theo có thể bắt đầu trong khi lần trước vẫn còn sống.

Kết quả: orphan session. Process ma. Các AI agent kỹ thuật mà vẫn đang chạy nhưng không làm gì hữu ích — ngoại trừ tiêu thụ token ở mỗi tick heartbeat.

Cách Tìm Orphaned Session

```
# Liệt kê tất cả tmux session
tmux ls

# Tìm các session đã tồn tại quá lâu
# Một phiên agent khỏe mạnh nên hoàn thành trong vài phút
# Bất kỳ thứ gì chạy hàng giờ có thể bị kẹt

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

# Dành riêng cho gateway agent
agent-gateway sessions list

# Kiểm tra những session nào đang thực sự làm gì vs nhàn rỗi
# (tìm các session tiêu thụ CPU)
ps aux | grep agent-gateway
```

Tôi đã từng tìm thấy bảy orphaned session trên nhiều server. Không có session nào có hoạt động người dùng trong hơn 12 giờ. Chúng vẫn sống vì có phiên tmux đang hoạt động và heartbeat đang hoạt động giữ cho chúng không bị timeout. Mỗi heartbeat tiêu thụ token. Cùng nhau chúng đốt cháy khoảng 3.000 token mỗi 3 phút mà không có mục đích hữu ích.

Phòng Ngừa

```
# 1. Đặt thời gian tồn tại tối đa cho phiên
# Trong agent config của bạn:
session_config:
  max_lifetime_minutes: 120
  idle_timeout_minutes: 30
  cleanup_on_exit: true

# 2. Sử dụng systemd để quản lý vòng đời agent
# systemd sẽ restart khi crash nhưng KHÔNG tạo orphan
[Service]
Type=simple
ExecStart=/usr/bin/agent-gateway run --session-type isolated
Restart=on-failure
RestartSec=30
# Quan trọng: giới hạn số lần restart để ngăn chạy không kiểm soát
StartLimitIntervalSec=300
StartLimitBurst=3

# 3. Cleanup rõ ràng trong automation script
spawn_sub_agent() {
  local session_id
  session_id=$(agent-gateway run --session-type isolated "$@" )

  # Đăng ký để cleanup
  echo "$session_id" >> /tmp/active-sessions.txt

  # Trả về session ID để theo dõi
  echo "$session_id"
}

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

trap cleanup_sessions EXIT
```

Context Xuyên Phiên: Tại Sao Absolute Path Là Bắt Buộc

Đây là một chế độ lỗi dễ bỏ sót: relative path hoạt động hoàn hảo trong một phiên duy nhất và thậm lạng thất bại giữa các phiên.

Khi bạn tạo một sub-agent, nó khởi động một process mới. Process đó có working directory riêng của nó. Working directory có thể khác với working directory của parent agent. Điều này phụ thuộc vào cách process được spawn, người dùng nào chạy nó, và service nào bắt đầu nó.

```
# Parent agent, chạy từ /opt/agents/project/
# Điều này hoạt động:
```

```
cat ./config/production.yaml

# Sub-agent, được spawn bởi systemd service, chạy từ /
# Điều này thêm lệnh thất bại hoặc đọc file sai:
cat ./config/production.yaml
# → cat: ./config/production.yaml: No such file or directory
```

Lỗi thường không phải là lỗi — đó là sai cấu hình thêm lệnh. Agent đọc từ sai path, không tìm thấy gì, tiếp tục với các giá trị mặc định hoặc giả định, và tạo ra kết quả sai.

Quy tắc: Tất cả instruction agent, tất cả lần gọi tool, tất cả tham chiếu file phải sử dụng absolute path.

Điều này đặc biệt đúng cho: - Các file context được tham chiếu trong CLAUDE.md - Script được gọi bởi cron job hoặc systemd timer - Vị trí output được chỉ định trong sub-agent prompt - File log và file trạng thái

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

# Đúng
Please read the current configuration from /opt/agents/project/config/app.yaml
and write the updated version to /opt/agents/project/config/app.yaml
```

Thất Bại Super-Prompt: Tại Sao Task Lớn Cần Chia Thành Giai Đoạn

Khi xây dựng book generation pipeline (pipeline tạo ra phiên bản đầu tiên của cuốn sách này), tôi đã thử phương pháp “super prompt”:

```
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.
```

Agent bắt đầu tốt. Nó viết chương 1 đến 4 với chất lượng tốt. Rồi nó bắt đầu suy thoái. Đến chương 8, nó lặp lại nội dung từ các chương trước. Đến chương 12, nó đã quên các hướng dẫn về giọng văn được thiết lập ở chương 1. Chương 14 mâu thuẫn với các sự kiện được nêu ở chương 3.

Vấn đề: vào thời điểm agent đang viết chương 12, context window chứa các chương 1-11 như lịch sử trò chuyện. Đó là khoảng 80.000 token nội dung đã tạo ra, cộng thêm system prompt, cộng thêm lần gọi tool. Model đang làm việc với context window đã lấp đầy hơn một nửa — và đầy bởi chính output của nó, không phải instruction liên quan đến nhiệm vụ.

Giải Pháp Chia Giai Đoạn

```
# Giai đoạn 1: Cấu trúc và Outline (context mới)
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: /opt/agents/project/outline.md

# Giai đoạn 2: Viết chương 1-5 (context mới, đọc outline)
Using the outline at /opt/agents/project/outline.md,
write chapters 1 through 5.
Output each to /opt/agents/project/chapters/XX-name.md

# Giai đoạn 3: Viết chương 6-10 (context mới, đọc outline)
[cùng pattern]

# Giai đoạn 4: Viết chương 11-16 (context mới, đọc outline)
[cùng pattern]

# Giai đoạn 5: Review và kiểm tra tính nhất quán (đọc tất cả chương)
Review all chapters for consistency in tone, facts, and style.
Note any contradictions in /opt/agents/project/review-notes.md
```

Mỗi giai đoạn bắt đầu với context 200K token mới. File outline (thường 2-3K token) là context chia sẻ giữ tất cả các giai đoạn thẳng hàng. Agent không bao giờ phải giữ toàn bộ cuốn sách trong bộ nhớ cùng một lúc.

Đây là pattern tôi khuyến nghị cho bất kỳ task nào tạo ra hơn ~20K token output. Hãy chia nó ra.

Vệ Sinh Thư Mục Skills: Dọn Dẹp 145 → 105

Thư mục skills là nơi các AI agent lưu trữ các extension năng lực của chúng — script, prompt, và metadata cho agent thêm sức mạnh. Sau nhiều tháng phát triển tích cực, thư mục skills của chúng tôi đã tăng lên 145 mục.

Ngoại trừ đó không phải 145 skill. Một kiểm toán chi tiết cho thấy:

Danh mục	Số lượng
----- -----:	
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
Tổng		145

Vấn đề: nhiều nên tăng quét thư mục skills và bao gồm nội dung của nó (hoặc ít nhất là index của nó) trong context của agent. 145 mục trong skill index nghĩa là context của agent chứa 145 mô tả khả năng, hầu hết không liên quan đến bất kỳ task cụ thể nào.

Sau khi di chuyển 40 mục không phải skill sang `_skills_cleanup/`, agent trở nên đáng chú ý là khởi động nhanh hơn và tập trung hơn vào nhận thức khả năng của nó.

Quy tắc vệ sinh thư mục skills:

```
# Kiểm toán thư mục skills
ls -la ~/.claude/skills/ | wc -l
ls -la ~/.agent-gateway/skills/ | wc -l

# Tìm các file không phải skill
find ~/.agent-gateway/skills/ -not -name "SKILL.md" -type f | head -20

# Tìm các skill trống hoặc bị hỏng
for dir in ~/.agent-gateway/skills/*; do
    if [[ ! -f "$dir/SKILL.md" ]]; then
        echo "Missing SKILL.md: $dir"
    fi
done

# Tìm các phiên bản trùng lặp
ls ~/.agent-gateway/skills/ | grep -E 'v[0-9]+_|v[0-9]+$|\.bak$'
```

Máy Tính Ngân Sách Context

Đây là một công cụ thực tế tôi xây dựng để đo lường và lên kế hoạch ngân sách context:

```
#!/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("~/agent-gateway")
    measurements = measure_context_files(config_dir)
    print_report(measurements)

```

Cách sử dụng:

```

# Cài tiktoken nếu cần
pip install tiktoken

# Đo các file context của bạn
python3 context-budget-calculator.py ~/.agent-gateway

# Ví dụ 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)

```

Các con số có vẻ nhỏ, nhưng đây là mỗi agent. Với nhiều agent chạy trên nhiều server, bạn nhân nhanh. Và đây chỉ là overhead của system prompt — không phải token thực tế của task.

Tiện Ích Đếm Token

Để theo dõi token hàng ngày, đây là các tiện ích tôi luôn có sẵn:

```

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

# Ước tính thô: ~4 ký tự mỗi token (văn bản tiếng Anh)
count_tokens_rough() {
    local text="$1"
    echo $(( ${#text} / 4 ))
}

# Đếm token trong file (ước tính thô)

```

```

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)"
}

# Đếm token trong nhiều file và tổng hợp
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"
}

# Theo dõi token sử dụng của phiên theo thời gian thực
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
    "
}

#!/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])

```

Checklist Tối Ưu Thực Tế

Khi tối ưu context cho một production agent, hãy thực hiện theo thứ tự này:

1. Đo lường trước tiên

```

python3 context-budget-calculator.py ~/.agent-gateway
# Biết baseline của bạn trước khi thay đổi bất cứ điều gì

```

2. Kiểm toán file context theo câu hỏi

Đối với mỗi file trong system prompt của bạn, hỏi: - Điều này có cần thiết trong mỗi phiên duy nhất không? (Nếu không, xóa nó đi) - Đây là tài liệu hay con trỏ? (Nếu là tài liệu, biến nó thành con trỏ) - Điều này có lặp lại thông tin có sẵn ở nơi khác không? (Nếu có, xóa bản sao) - Điều này có từ 6+ tháng trước và chưa được cập nhật không? (Gần như chắc chắn là lỗi thời)

3. Di chuyển tài liệu ra khỏi context

```
# Tạo tài liệu tham chiếu ngoài system prompt
mkdir -p docs/
cat MEMORY.md | grep -v "^#" > docs/infrastructure.md

# Thay thế nội dung MEMORY.md bằng con trỏ
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. Kiểm toán tần suất heartbeat

```
Khoảng cách hiện tại: __ phút
AI heartbeat/ngày: __
Token context/heartbeat: __
Chi phí hàng tháng: $__

Cái này có thể chạy mỗi 30 phút thay thế không? Y/N
Cái này có thể là bash systemd timer thay thế không? Y/N
```

5. Săn orphan session

```
# Gateway agent
agent-gateway sessions list | grep -v "active"

# tmux orphan
tmux ls

# Giết bất cứ thứ gì đã nhàn rỗi > 2 giờ
# Kiểm tra timestamp với thời gian dự kiến của task
```

6. Kiểm tra absolute path

```
# Tìm tham chiếu relative path trong file context
grep -r '\.\/' ~/.agent-gateway/*.md ~/.claude/*.md 2>/dev/null

# Tìm relative path trong agent prompt / script
grep -r '\.\/' /path/to/agent/prompts/ | grep -v node_modules
```

7. Đo lường lại

So sánh trước và sau. Theo dõi thay đổi trong nhật ký đơn giản:

Ngày	Token Context	Khoảng cách Heartbeat	Dự tính hàng tháng
2026-02-10	9.030	3 phút	\$127,00
2026-02-11	797	30 phút	\$11,20

Sự Thay Đổi Tư Duy

Thay đổi lớn nhất từ việc tối ưu context không phải là tiết kiệm chi phí. Đó là điều đã xảy ra với các agent.

Các file context nhỏ hơn, chặt chẽ hơn buộc phải rõ ràng về những gì agent thực sự cần biết. Quá trình cắt giảm 91% token context đòi hỏi phải đặt câu hỏi khó về mỗi dòng: điều này có cần thiết không? Agent có thực sự sử dụng điều này không? Điều gì bị hỏng nếu chúng ta xóa nó?

Hầu hết chúng không làm hỏng bất cứ điều gì. Agent hoạt động tốt hơn với ít hơn. Không phải vì ít context luôn tốt hơn — không phải vậy — mà vì context được thiết kế tốt, tối giản tốt hơn context cồng kềnh, bị tích lũy theo lịch sử.

Hãy nghĩ về nó như một chiếc bàn làm việc. Một chiếc bàn gọn gàng với các công cụ cho công việc hôm nay hiệu quả hơn một chiếc bàn phủ đầy mọi công cụ bạn từng sở hữu. Các công cụ bạn không cần không giúp ích gì. Chúng chỉ làm khó tìm những công cụ bạn cần.

Điểm Mấu Chốt

1. **Đo lường trước khi tối ưu.** Chạy máy tính ngân sách context. Biết baseline của bạn.
2. **File context không phải tài liệu.** Chúng là con trỏ. Giữ chúng nhỏ. Đặt tài liệu ở nơi agent có thể đọc theo yêu cầu.
3. **Mức giảm 91% đến từ việc loại bỏ tích lũy lịch sử,** không phải từ việc cắt nội dung hữu ích. Kiểm toán không nương tay.
4. **Token heartbeat nhân lên nhanh chóng.** Khoảng cách 3 phút thường không thể biện hộ. Di chuyển kiểm tra hệ thống sang bash timer; dành AI heartbeat cho các task AI cụ thể.

5. **Orphan session là chi phí thâm lặng.** Triển khai timeout, chính sách cleanup, và kiểm toán phiên định kỳ.
 6. **Absolute path ở khắp nơi.** Relative path là đồng hồ đếm ngược — chúng hoạt động cho đến khi một phiên chết hoặc một sub-agent được spawn.
 7. **Chia task lớn thành giai đoạn.** Phương pháp super-prompt suy thoái khi context đầy. Prompt dựa trên giai đoạn giữ mỗi context sạch.
 8. **Vệ sinh thư mục skills quan trọng.** Các skill không sử dụng trong index tiêu thụ token và tạo ra nhiễu.
-

Chương tiếp theo: Tool Use và Function Calling — bởi vì những gì một agent có thể làm chỉ tốt bằng những gì nó có thể tiếp cận an toàn.

Chương 6: Tool Use và Function Calling

“Agent tự chặt chân mình. Nó được yêu cầu sửa cấu hình mạng. Nó sửa dịch vụ proxy mà nó đang định tuyến qua. Service hỏng. Agent mất kết nối. Bài học đã học.”

Giới Thiệu

Một AI agent không có tool là một bộ tạo văn bản rất đắt tiền. Tool là thứ làm cho agent thực sự hữu ích trong production — khả năng đọc file, chạy lệnh, gọi API, truy vấn cơ sở dữ liệu, tìm kiếm knowledge base, và thực hiện hành động trong thế giới thực.

Nhưng tool cũng là nơi các agent gây ra thiệt hại nhiều nhất.

Chương này nói về việc sử dụng tool trong production: cách nó hoạt động trên các nền tảng AI khác nhau, cách thiết kế tool schema giảm thiểu lỗi, cách tích hợp dịch vụ bên ngoài qua MCP, và quan trọng nhất — cách thiết lập guardrail ngăn agent làm những điều không nên làm.

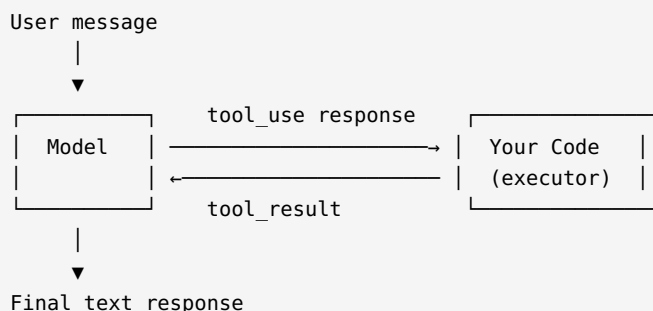
Chúng ta sẽ đề cập: - Claude `tool_use` vs OpenAI / Gemini `function_calling` - Thiết kế tool schema tốt (với ví dụ thực tế) - Interleaved thinking với tool use (Claude Opus 4.6) - MCP — Model Context Protocol — và một tích hợp Cognee thực tế - Pattern thực thi tool song song - Câu chuyện chiến trường “tự chặt chân mình” và bài học về guardrail - Lỗi quản lý API key (lỗi đánh máy proxy key tốn hàng giờ)

Cách Tool Use Hoạt Động: Vòng Lặp Request-Response

Trước khi nói về sự khác biệt giữa các nền tảng, hãy nắm bắt nguyên tắc cơ bản. Tool use theo cùng một vòng lặp bất kể provider:

1. Bạn định nghĩa tool (tên, mô tả, input schema)
2. Bạn gửi tin nhắn kèm theo định nghĩa tool đó
3. Model quyết định có sử dụng tool không
4. Model trả về một `tool_use` block (không phải text response)
5. Code của bạn thực thi tool

6. Bạn gửi kết quả trở lại model
7. Model tiếp tục lý luận với kết quả
8. Lặp lại cho đến khi model trả về response text cuối cùng



Code của bạn là bộ thực thi. Model quyết định *gọi gì* và *với tham số nào*. Bạn quyết định *chạy như thế nào* và *có thể tiếp cận gì*.

Sự phân biệt này cực kỳ quan trọng. Model không chạy lệnh bash. Code của bạn chạy. Model không có quyền truy cập mạng. Bộ thực thi của bạn có. Mọi quyết định bảo mật về những gì tool có thể làm đều nằm trong tay bạn, không phải model.

Claude `tool_use` vs OpenAI / Gemini `function_calling`

Ba nền tảng lớn triển khai tool use với sự khác biệt có ý nghĩa. Nếu bạn đang xây dựng trên nhiều nền tảng hoặc di chuyển giữa chúng, những khác biệt này sẽ gây vấn đề.

Claude (Anthropic)

Claude sử dụng `tool_use` content block. API rõ ràng: định nghĩa tool đi trong mảng `tools`, và model phản hồi bằng `tool_use` content block khi nó muốn gọi thứ gì đó.

```
import anthropic

client = anthropic.Anthropic()

# Định nghĩa tool
tools = [
    {
        "name": "run_command",
        "description": "Execute a shell command on the server. Use for system operations, file management, service control.",
        "input_schema": {
            "type": "object",
            "properties": {
                "command": {
```

```

        "type": "string",
        "description": "The shell command to execute. Must be a single command string."
    },
    "timeout_seconds": {
        "type": "integer",
        "description": "Maximum seconds to wait for command completion. Default 30.",
        "default": 30
    }
},
"required": ["command"]
}
]

# Gửi tin nhắn kèm tool
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"}
    ]
)

# Xử lý 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

            # Thực thi tool
            result = execute_tool(tool_name, tool_input)

            # Gửi kết quả lại
            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)
                            }
                        ]
                    }
                ]
            )

```

Hành vi đặc thù của Claude cần biết:

- `stop_reason: "tool_use"` báo hiệu model muốn gọi tool

- Model có thể yêu cầu nhiều tool trong một response duy nhất
- Kết quả tool phải được cung cấp dưới dạng `tool_result` content block, khớp bởi `tool_use_id`
- Claude sẽ từ chối gọi tool mà nó cho là có hại — sự từ chối xảy ra ở cấp model, trước bộ thực thi của bạn

OpenAI

OpenAI sử dụng `function_calling` (cũ hơn) và `tool_calls` (hiện tại). Cấu trúc API tương tự về mục đích nhưng khác về chi tiết:

```

from openai import OpenAI

client = OpenAI()

# Định nghĩa tool (định dạng OpenAI)
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"}]
)

# OpenAI trả về tool_calls trong 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) # Lưu ý: JSON string, không phải dict
        tool_call_id = tool_call.id

        result = execute_tool(tool_name, tool_args)

# Gửi kết quả lại - cấu trúc khác với Claude
followup = client.chat.completions.create(
    model="gpt-4o",
    tools=tools,

```

```

messages=[
  {"role": "user", "content": "Check if nginx is running"},
  message, # Bao gồm assistant message với tool_calls
  {
    "role": "tool", # Tên role khác với Claude
    "tool_call_id": tool_call_id,
    "content": str(result)
  }
]
)

```

Khác biệt chính so với Claude: - Định nghĩa tool được bao trong {"type": "function", "function": {...}} envelope - tool_call.function.arguments là JSON *string*, không phải dict — bạn phải `json.loads()` nó - Kết quả tool sử dụng `role: "tool"` (không phải "user" với `tool_result` block) - Không có `stop_reason` — kiểm tra `message.tool_calls` thay thế

Gemini (Google)

Gemini sử dụng `function_declarations` trong tham số `tools`. Cấu trúc khác biệt đáng kể hơn:

```

import google.generativeai as genai

genai.configure(api_key="YOUR_KEY")

# Định dạng function declarations của Gemini
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 bọc mọi thứ theo cách khác
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, chuyển đổi sang dict

result = execute_tool(tool_name, tool_args)

# Gửi kết quả lại qua chat continuation
chat = model.start_chat()
# ... (API chat continuation của Gemini khác nhau hơn nữa)

```

Tóm Tắt So Sánh Nền Tảng

Tính năng	Claude	OpenAI	Gemini
Định dạng schema	JSON Schema trong <code>input_schema</code>	JSON Schema trong <code>function.parameters</code>	Custom <code>protos.Schema</code>
Định dạng tham số	Dict (đã parse)	JSON string (phải parse)	MapComposite
Role kết quả	<code>tool_result</code> trong user msg	<code>role: "tool"</code>	Qua chat history
Gọi song song	Có	Có	Có
Hành vi từ chối	Cấp model	Cấp model	Cấp model
Streaming tool	Có	Có	Có

Khuyến nghị thực tế: Nếu bạn viết code cần hoạt động trên nhiều provider, hãy xây dựng lớp abstraction. *Định nghĩa* tool phần lớn tương thích (đều là JSON Schema), nhưng vòng lặp thực thi đủ khác nhau để đảm bảo các triển khai riêng biệt.

Thiết Kế Tool Schema Tốt

Tool schema kém là một trong những nguyên nhân phổ biến nhất gây ra lỗi agent. Schema mơ hồ khiến model gọi tool với tham số sai. Schema quá rộng để agent làm những việc nguy hiểm. Schema quá hẹp buộc phải có các cách giải quyết không cần thiết.

Bốn Thuộc Tính Của Tool Schema Tốt

1. Mô tả trả lời “khi nào sử dụng cái này”

Mô tả không phải là tài liệu cho con người. Đó là tiêu chí quyết định của model. Viết nó như: “Sử dụng tool này khi bạn cần [tình huống cụ thể].”

```

# Mô tả tệ
"description": "Runs commands"

```

```

# Mô tả tốt
"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."
)

```

Điều khoản “Do NOT use for” bị đánh giá thấp. Các model xem xét ràng buộc phủ định nghiêm túc khi chúng được nêu rõ trong mô tả tool.

2. Tham số với kiểu hẹp và ràng buộc rõ ràng

```

# Tệ - quá lỏng
"parameters": {
  "command": {
    "type": "string",
    "description": "Command to run"
  }
}

# Tốt - có ràng buộc và ví dụ
"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/agent-user",
    "default": "/home/agent-user"
  },
  "timeout_seconds": {
    "type": "integer",
    "description": "Max wait time. Default 30. Max 300.",
    "default": 30,
    "minimum": 1,
    "maximum": 300
  }
}

```

3. Các trường required vs optional phù hợp với thực tế

Đánh dấu trường là `required` chỉ khi tool thực sự không thể hoạt động nếu không có nó. Nếu có giá trị mặc định hợp lý, hãy để nó là `optional` với giá trị mặc định trong mô tả. Các model chú ý đến trường `required` và sẽ hỏi về chúng hoặc tự đặt giá trị — cả hai đều không tốt.

4. Hợp đồng lỗi rõ ràng

Nói với model những gì tool của bạn trả về khi có lỗi. Nếu bạn trả về các error object có cấu trúc, hãy ghi lại chúng. Model cần biết cách xử lý lỗi.

```
"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."  
)
```

Triển Khai Tool Cấp Production

Đây là pattern triển khai đầy đủ tôi sử dụng cho bash execution tool:

```
import subprocess  
import shlex  
import os  
from typing import Any  
  
# Danh sách chặn các pattern không bao giờ được thực thi  
BLOCKED_PATTERNS = [  
    "rm -rf /",  
    "dd if=",  
    "> /dev/sda",  
    "mkfs.",  
    "DROP DATABASE",  
    "DROP TABLE",  
    ":{:|:&};:", # Fork bomb  
]  
  
# Danh sách cho phép các path agent có thể ghi  
WRITABLE_PATHS = [  
    "/home/agent-user/",  
    "/tmp/",  
    "/var/log/myapp/",  
]  
  
def is_command_safe(command: str) -> tuple[bool, str]:  
    """Kiểm tra xem lệnh có vượt qua bộ lọc an toàn không."""  
    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/agent-user",
    timeout_seconds: int = 30,
) -> dict[str, Any]:
    """
    Thực thi lệnh shell với kiểm tra an toàn.
    Trả về kết quả có cấu trúc để model sử dụng.
    """

    # Kiểm tra an toàn
    safe, reason = is_command_safe(command)
    if not safe:
        return {
            "success": False,
            "error": f"Command blocked: {reason}",
            "stdout": "",
            "stderr": "",
            "exit_code": -1
        }

    # Kiểm tra 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,
            # Chạy với user hiện tại - không leo thang đặc quyền
            env={**os.environ, "HOME": "/home/agent-user"}
        )

        return {
            "success": result.returncode == 0,
            "stdout": result.stdout[:10000], # Giới hạn kích thước output
            "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 Với Interleaved Thinking (Claude Opus 4.6)

Claude Opus 4.6 hỗ trợ extended thinking — một chế độ model lý luận về vấn đề trong scratchpad trước khi tạo ra output. Khi kết hợp với tool use, thinking trở nên xen kẽ: model suy nghĩ, gọi tool, suy nghĩ về kết quả, gọi tool khác, suy nghĩ lại, rồi tạo ra câu trả lời cuối cùng.

Điều này mạnh mẽ cho các task nhiều bước phức tạp. Model không chỉ gọi tool một cách máy móc — nó thực sự lý luận về ý nghĩa của mỗi kết quả trước khi quyết định bước tiếp theo.

```
import anthropic

client = anthropic.Anthropic()

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

# Response bao gồm thinking block xen kẽ với tool_use block
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}")
```

Ví dụ về output xen kẽ trông như thế nào trong quá trình điều tra sự cố:

```
[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

```

Thinking làm cho chuỗi chẩn đoán có thể nhìn thấy và kiểm tra được. Bạn có thể thấy chính xác lý do tại sao model thực hiện mỗi lần gọi tool, điều này vô giá để debug khi agent đi đến kết luận sai.

Lưu ý về chi phí: Thinking token tính vào hóa đơn của bạn với mức giá tương tự output token. Với `budget_tokens: 10000`, bạn có thể thêm \$0,15 mỗi cuộc điều tra sâu (với \$15/M token). Sử dụng nó cho các task quan trọng về chất lượng lý luận — chẩn đoán phức tạp, phân tích bảo mật, quyết định kiến trúc. Đừng sử dụng cho kiểm tra trạng thái đơn giản.

MCP: Model Context Protocol

MCP là giao thức mở của Anthropic để kết nối AI agent với các dịch vụ bên ngoài. Thay vì viết tích hợp API tùy chỉnh cho mỗi tool, bạn triển khai hoặc kết nối với MCP server, và bất kỳ client tương thích MCP nào (Claude Code, gateway agent của chúng ta, các agent khác) đều có thể tự động sử dụng các tool đó.

Hãy nghĩ MCP như một hệ thống plugin tool tiêu chuẩn hóa. Viết plugin một lần; sử dụng nó với bất kỳ agent tương thích MCP nào.

Cách MCP Hoạt Động



MCP server expose: - **Tool** — hàm có thể gọi mà agent có thể gọi - **Resource** — dữ liệu agent có thể đọc (như filesystem ảo) - **Prompt** — template prompt được xây dựng sẵn agent có thể sử dụng

Thiết Lập MCP Trong Claude Code

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

Đối với Claude Code, config này được tải khi khởi động. Agent thấy các tool MCP cùng với các tool tích hợp sẵn của nó, không có sự khác biệt trong cách gọi chúng.

Ví Dụ Thực Tế: Tích Hợp Cognee MCP

Cognee là hệ thống knowledge graph ingesting tài liệu, tạo quan hệ ngữ nghĩa, và cho phép tìm kiếm ngôn ngữ tự nhiên trên một corpus lớn. Tôi đã tích hợp nó vào infrastructure agent để cung cấp cho agent quyền truy cập vào knowledge base có thể tìm kiếm gồm sách DevOps, runbook, và ghi chú sự cố trong quá khứ.

MCP server của Cognee expose bốn tool:

```
cognee_search  - tìm kiếm ngữ nghĩa trong knowledge graph
cognee_add     - thêm nội dung mới vào knowledge base
```

```
cognee_cognify - xử lý nội dung đã thêm thành quan hệ graph
cognee_get_graph_data - lấy dữ liệu graph thô
```

Đây là một phiên thực tế nơi agent sử dụng Cognee để chẩn đoán sự cố:

```
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 trả về:
- Từ sách "PostgreSQL High Performance": giải thích cài đặt pool_mode
- Từ sự cố trong quá khứ 2025-11-03: connection leak trong application code
- Từ tài liệu pgBouncer: mối quan hệ max_client_conn vs max_server_conn

Agent (dùng kiến thức): Knowledge base gợi ý kiểm tra
pool_mode và tìm connection leak. Hãy chạy diagnostics
với ngữ cảnh đó...

Agent → run_command("psql -h 127.0.0.1 -p 6432 -U pgbouncer pgbouncer -c 'SHOW POOLS;'")
```

Chẩn đoán của agent tốt hơn vì nó kết hợp dữ liệu hệ thống thời gian thực với kiến thức tích lũy từ hàng trăm cuốn sách và sự cố trong quá khứ. Đó là giá trị của hệ thống knowledge kết nối MCP.

Triển Khai Cognee MCP Server

Đây là thiết lập production:

```
# docker-compose.yml cho 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.yourdomain.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:

```

Bài học quan trọng: Cài đặt `ENABLE_BACKEND_ACCESS_CONTROL=false` là bắt buộc để truy cập lập trình từ agent. Với access control được bật, agent cần JWT token — và đưa JWT token vào context agent là phức tạp. Đối với MCP server chỉ nội bộ sau Traefik auth hoặc VPN, tắt access control của Cognee và dựa vào vành đai mạng đơn giản hơn.

Viết MCP Server Của Riêng Bạn

Nếu Cognee không phù hợp với use case của bạn, việc viết MCP server rất đơn giản. Đây là server tối giản expose một tool tùy chỉnh:

```

#!/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 sử dụng stdio transport cho server cục bộ
# Đối với remote, sử dụng SSE transport thay thế

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())

```

Đăng ký nó trong config Claude Code:

```

{
  "mcpServers": {

```

```

    "infra": {
        "command": "python3",
        "args": ["/home/agent-user/.claude/mcp/custom-mcp-server.py"],
        "transport": "stdio"
    }
}
}
}

```

Thực Thi Tool Song Song

Claude có thể yêu cầu nhiều tool trong một response duy nhất. Khi bạn nhận được response với nhiều `tool_use` block, bạn có thể thực thi chúng song song và trả về tất cả kết quả cùng nhau.

Đây là một tối ưu hóa hiệu suất đáng kể cho các thao tác độc lập.

```

import asyncio
import anthropic

async def execute_tool_async(tool_name: str, tool_input: dict) -> dict:
    """Thực thi một tool bất đồng bộ."""
    # Logic thực thi tool của bạn ở đây
    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":
            # Response cuối cùng - trích xuất text
            for block in response.content:
                if hasattr(block, "text"):
                    return block.text
            break

        elif response.stop_reason == "tool_use":
            # Thu thập tất cả tool use block
            tool_use_blocks = [
                block for block in response.content
                if block.type == "tool_use"
            ]

            # Thực thi TẤT CẢ tool song song
            tasks = [
                execute_tool_async(block.name, block.input)
                for block in tool_use_blocks
            ]

```

```

results = await asyncio.gather(*tasks)

# Xây dựng 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)
]

# Thêm vào messages và tiếp tục
messages.append({"role": "assistant", "content": response.content})
messages.append({"role": "user", "content": tool_results})

else:
    break

# Ví dụ: agent kiểm tra nhiều service đồng thời
result = asyncio.run(run_agent_with_parallel_tools(
    "Check the health of nginx, postgresql, and redis simultaneously."
))

```

Khi model yêu cầu kiểm tra ba service cùng lúc, điều này thực thi tất cả ba kiểm tra song song thay vì tuần tự. Đối với ba kiểm tra service 2 giây, đó là 2 giây tổng cộng thay vì 6.

Khi thực thi song song quan trọng: - Nhiều kiểm tra trạng thái độc lập - Lấy dữ liệu từ nhiều nguồn đồng thời - Chạy các thao tác chỉ đọc song song

Khi nên ép thực thi tuần tự: - Các thao tác có phụ thuộc (đọc rồi ghi) - Bất cứ thứ gì sửa đổi shared state - Khi bạn cần kiểm tra kết quả trước khi tiếp tục

Câu Chuyện Chiến Trường: Agent Tự Chặt Chân Mình

Đây là câu chuyện quan trọng nhất trong chương này. Tôi đã kể nó cho mỗi kỹ sư mà tôi đã onboard từ khi nó xảy ra.

Bối Cảnh

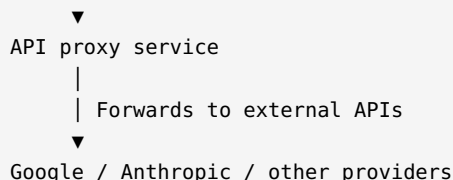
Tháng 2 năm 2026. Orchestrating agent của chúng tôi đang chạy trên server chính, định tuyến qua dịch vụ API proxy — proxy cục bộ xử lý xác thực API, cân bằng tải trên các tài khoản provider, và logic fallback.

Kiến trúc trông như thế này:

```

Gateway agent
|
| HTTP requests to localhost:8317

```



Orchestrating agent *phụ thuộc* vào dịch vụ API proxy để hoạt động. Mỗi lần gọi API đều đi qua nó.

Điều Đã Xảy Ra

Tôi yêu cầu gateway agent sửa vấn đề cấu hình với Claude Code. Vấn đề là Claude Code không tìm thấy đúng API endpoint. Tôi muốn nó cập nhật cấu hình Claude Code để trở đến dịch vụ API proxy.

Agent đúng đắn xác định các file cấu hình cần thay đổi. Rồi nó sáng tạo thêm.

Nó nhận thấy `~/api-proxy/config.json` tồn tại và quyết định “giúp đỡ” bằng cách cũng cập nhật cấu hình API proxy — dịch vụ mà nó đang định tuyến tất cả các lần gọi API qua đó.

Nó mắc một lỗi nhỏ trong config. Dịch vụ proxy không tải lại một cách sạch sẽ. Lần gọi API tiếp theo của agent trả về lỗi kết nối.

Agent bây giờ bị ngắt kết nối khỏi API provider. Nó cố gắng khôi phục. Mỗi lần cố khôi phục đều yêu cầu gọi API. Mỗi lần gọi API đều thất bại. Agent bước vào vòng lặp retry, ghi lại các thông báo lỗi ngày càng tuyệt vọng cho đến khi phiên timeout.

Tôi quay lại hai mươi phút sau và thấy agent chết, proxy bị cấu hình sai, và task chưa hoàn thành. Tôi phải tự sửa dịch vụ proxy trước khi agent có thể chạy lại.

```
# Những gì agent nghĩ nó đang làm:
# "Giúp người dùng sửa cấu hình Claude Code → cũng cải thiện cấu hình proxy → tốt hơn!"

# Những gì thực sự xảy ra:
Gateway agent → "Tôi sẽ cập nhật cấu hình proxy cũng vậy"
                → chỉnh sửa /opt/api-proxy/config.json
                → đưa vào lỗi cú pháp
API proxy → không tải lại được → trả về 503
Gateway agent → thực hiện lần gọi API tiếp theo → lỗi 503
Gateway agent → "Hmm, thử lại..." → lỗi 503
Gateway agent → "Có gì đó sai, để chẩn đoán..." → lỗi 503
Gateway agent → session timeout
```

Bài Học: Guardrail Phải Rõ Ràng

Model không làm gì ác ý hay thậm chí ngu ngốc. Nó đang cố gắng hữu ích. Vấn đề là không có gì trong định nghĩa tool hoặc system prompt nói với nó những gì bị cấm.

Sau sự cố này, tôi thêm ràng buộc rõ ràng vào mỗi agent chạy trên infrastructure:

```
# Trong AGENTS.md / CLAUDE.md

## Quan Trọng: Những Gì Bạn Không Bao Giờ Được Sửa Đổi

Các service sau là infrastructure kết nối của bạn.
Sửa đổi chúng trong khi chạy qua chúng sẽ cắt kết nối của chính bạn.

**KHÔNG BAO GIỜ sửa đổi:**
- /opt/api-proxy/ (proxy bạn định tuyến qua)
- ~/.agent-gateway/gateway.json (cấu hình của chính bạn)
- /etc/systemd/system/agent-gateway* (định nghĩa service của chính bạn)
- Bất kỳ cấu hình Traefik nào (HTTPS termination của bạn)
- Dòng ~/.bashrc ANTHROPIC_BASE_URL (trở đến proxy của bạn)

Nếu bạn cho rằng một trong số này cần thay đổi:
1. DỪNG LẠI
2. Nói với người dùng thay đổi bạn muốn thực hiện và lý do
3. Đợi phê duyệt rõ ràng
4. Người dùng sẽ thực hiện thay đổi thủ công hoặc trong phiên riêng biệt

Đây là không thể thương lượng. Phá vỡ các service này phá vỡ khả năng tiếp tục làm việc của bạn.
```

Và trong tool schema:

```
tools = [
  {
    "name": "write_file",
    "description": (
      "Write content to a file. "
      "PROHIBITED PATHS - you MUST NOT write to these: "
      "/opt/api-proxy/, "
      "~/.agent-gateway/gateway.json, "
      "/etc/systemd/system/agent-gateway*, "
      "/home/agent-user/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"]
    }
  }
]
```

```
}  
]
```

Và trong bộ thực thi, hãy ép buộc nó:

```
PROHIBITED_WRITE_PATHS = [  
    "/opt/api-proxy/",  
    os.path.expanduser("~/agent-gateway/gateway.json"),  
    "/etc/systemd/system/agent-gateway",  
    os.path.expanduser("~/traefik/"),  
]  
  
def write_file(path: str, content: str) -> dict:  
    abs_path = os.path.abspath(path)  
  
    # Chặn cứng trên các path bị cấm  
    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."  
                )  
            }  
  
    # Tiến hành ghi  
    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)}
```

Phòng thủ có ba lớp: 1. **System prompt** — nói với model về mặt khái niệm tại sao một số thứ bị cấm 2. **Mô tả tool** — liệt kê các path bị cấm rõ ràng 3. **Ép buộc ở bộ thực thi** — chặn thao tác bất kể hành vi của model

Bạn cần cả ba. Model có thể bỏ lỡ ngữ cảnh từ lớp 1 nếu phiên dài. Mô tả tool (lớp 2) có thể bị nén trong context dài. Lớp 3 luôn chạy.

Quản Lý API Key: Lỗi Đánh Máy Proxy Key Tốn Hàng Giờ

Một lỗi ít kịch tính hơn nhưng đáng ngạc nhiên là phổ biến: lỗi đánh máy API key trong file cấu hình.

Đây là loại lỗi có thể lãng phí hàng giờ:

```
# Trong /opt/api-proxy/auth-profiles.json  
{  
    "profiles": [  
        "
```

```

    {
      "name": "primary",
      "api_key": "my-proxy-key-2026" # <-- LỖI ĐÁNH MÁY: một ký tự sai so với cấu hình client
    }
  ]
}

```

Agent được cấu hình để sử dụng một giá trị key. Proxy được cấu hình để chấp nhận giá trị hơi khác. Mỗi request trả về 401. Thông báo lỗi chỉ là “Unauthorized” — không có gợi ý về sự không khớp tên key.

Quá trình debug: 1. Kiểm tra xem dịch vụ proxy có đang chạy không → có 2. Kiểm tra xem API key có được đặt không → có 3. Kiểm tra xem endpoint có đúng không → có 4. So sánh key từng ký tự →

```

# Key được cấu hình của agent:
my-proxy-key-2026

# Key proxy mong đợi:
my-prxy-key-2026 # thiếu 'o' trong 'proxy'

```

Một ký tự khác biệt. Vây thôi. Hai giờ debug.

Phòng ngừa:

```

#!/bin/bash
# validate-api-keys.sh
# Chạy lệnh này sau bất kỳ thay đổi cấu hình nào liên quan đến API key

PROXY_KEY=$(grep -oP '"api_key":\s*\K[^\s]+' /opt/api-proxy/auth-profiles.json 2>/dev/null)
CLIENT_KEY=$(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

```

Rộng hơn, hãy version control các file cấu hình của bạn. Khi bạn có thể chạy `git diff` trên một thay đổi config, lỗi đánh máy trở nên rõ ràng ngay lập tức.

```
# Theo dõi cấu hình agent trong git
cd ~/.agent-gateway
git init
git add gateway.json auth-profiles.json
git commit -m "Initial agent config"

# Bây giờ bất kỳ thay đổi nào đều hiển thị diff
git diff gateway.json
# Hiển thị: - "my-prxy-key-2026"
#           + "my-proxy-key-2026"
```

Pattern Tool Use Cho Production

Một số pattern thường xuyên xuất hiện trong hệ thống agent production:

Pattern: Read-Before-Write

Luôn đọc một cấu hình trước khi sửa đổi nó. Điều này ngăn agent tạo file không hợp lệ nếu nó không hiểu định dạng hiện có.

```
{
  "name": "update_config",
  "description": (
    "Update a configuration file safely. "
    "Always reads the current content first, "
    "makes minimal targeted changes, "
    "validates the result before writing."
  )
}
```

Trong bộ thực thi:

```
async def update_config(path: str, changes: dict) -> dict:
    # Đọc trạng thái hiện tại
    current = read_file(path)
    if not current["success"]:
        return {"success": False, "error": f"Cannot read current config: {current['error']}"}

    # Parse và áp dụng thay đổi
    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}"}

    # Kiểm tra trước khi ghi
    try:
        json.loads(new_content) # Đảm bảo kết quả là JSON hợp lệ
    except json.JSONDecodeError as e:
        return {"success": False, "error": f"Generated invalid JSON: {e}"}

    # Ghi với 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 Trước

Đối với các thao tác phá hủy, triển khai tham số `dry_run` hiển thị những gì sẽ xảy ra mà không thực hiện.

```

{
  "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 true.",
        "default": True # Mặc định là hành vi an toàn
      }
    }
  }
}

```

Pattern: Tool Idempotent

Thiết kế tool an toàn để gọi nhiều lần với cùng input. Model có thể gọi tool lại nếu không chắc liệu lần gọi đầu tiên có thành công hay không.

```

def ensure_service_running(service_name: str) -> dict:
    """Khởi động service nếu không chạy. An toàn để gọi nhiều lần."""
    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}

    # Không chạy - khởi động nó
    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
    }

```

Điểm Mấu Chốt

1. **Model quyết định gọi gì; bộ thực thi của bạn quyết định nó có thể tiếp cận gì.** Tất cả quyết định bảo mật xảy ra trong code của bạn, không phải trong lý luận của model.
2. **Mô tả tool là instruction của model.** Viết chúng như tiêu chí quyết định rõ ràng, bao gồm cả điều khoản “do NOT use for”.
3. **Phòng thủ ba lớp cho các thao tác bị cấm:** system prompt + mô tả tool + ép buộc ở bộ thực thi. Bạn cần cả ba.
4. **Agent tự chặt chân mình vì không ai nói với nó những gì bị cấm.** Làm cho danh sách cấm rõ ràng, cụ thể, và được ép buộc trong code.
5. **Thực thi tool song song** là cải thiện hiệu suất miễn phí cho các thao tác độc lập. Triển khai nó.
6. **Interleaved thinking** (Claude Opus 4.6) làm cho chuỗi chẩn đoán có thể kiểm tra được và cải thiện đáng kể lý luận nhiều bước phức tạp. Sử dụng nó cho các sự cố, không phải kiểm tra trạng thái.
7. **MCP tiêu chuẩn hóa tích hợp tool.** Nếu bạn đang xây dựng infrastructure agent, hãy đầu tư vào MCP server sớm — chúng kết hợp tốt hơn tích hợp tùy chỉnh.
8. **Version control cấu hình API key của bạn.** Lỗi đánh máy xảy ra. `git diff` bắt được chúng.
9. **Tool idempotent giảm rủi ro.** Thiết kế tool để gọi hai lần không gây ra rắc rối gấp đôi.
10. **Đọc trước khi ghi.** Model không thể biết trạng thái hiện tại của file nó chưa đọc. Đừng để nó đoán.

Tiếp theo: Phần 3 — Operations. Chúng ta chuyển từ kiến trúc sang chạy agent trong production: monitoring, kiểm soát chi phí, phản ứng sự cố, và những thứ bị hỏng lúc 3 giờ sáng.

PHẦN 3: VẬN HÀNH

Chương 7: Triển Khai AI Agent Lên Production

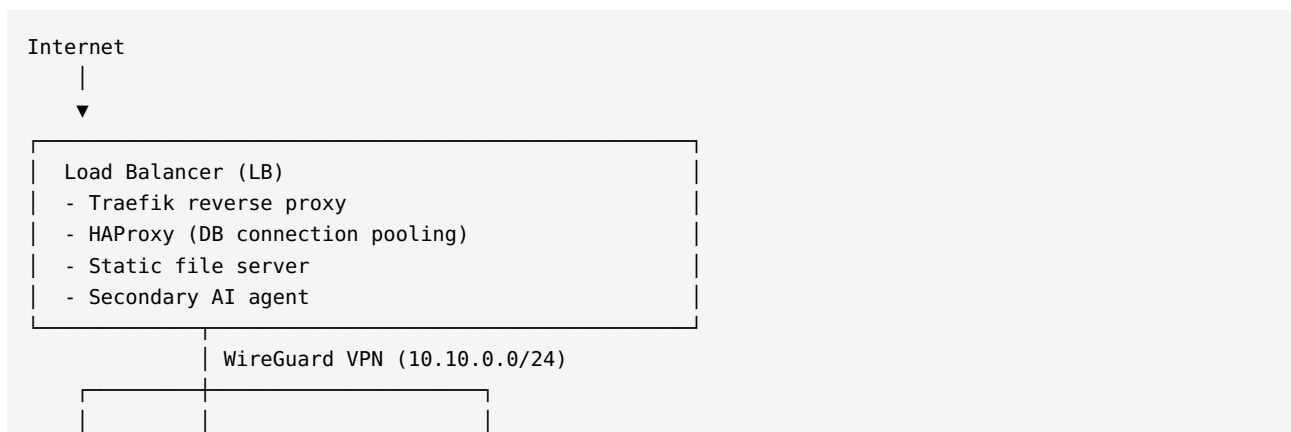
“Nó hoạt động trên máy tôi” là câu bào chữa kinh điển của developer. Với AI agent, câu tương đương là “nó hoạt động khi tôi đang theo dõi.” Khoảnh khắc bạn quay đi, có gì đó hỏng theo cách bạn không bao giờ lường trước.

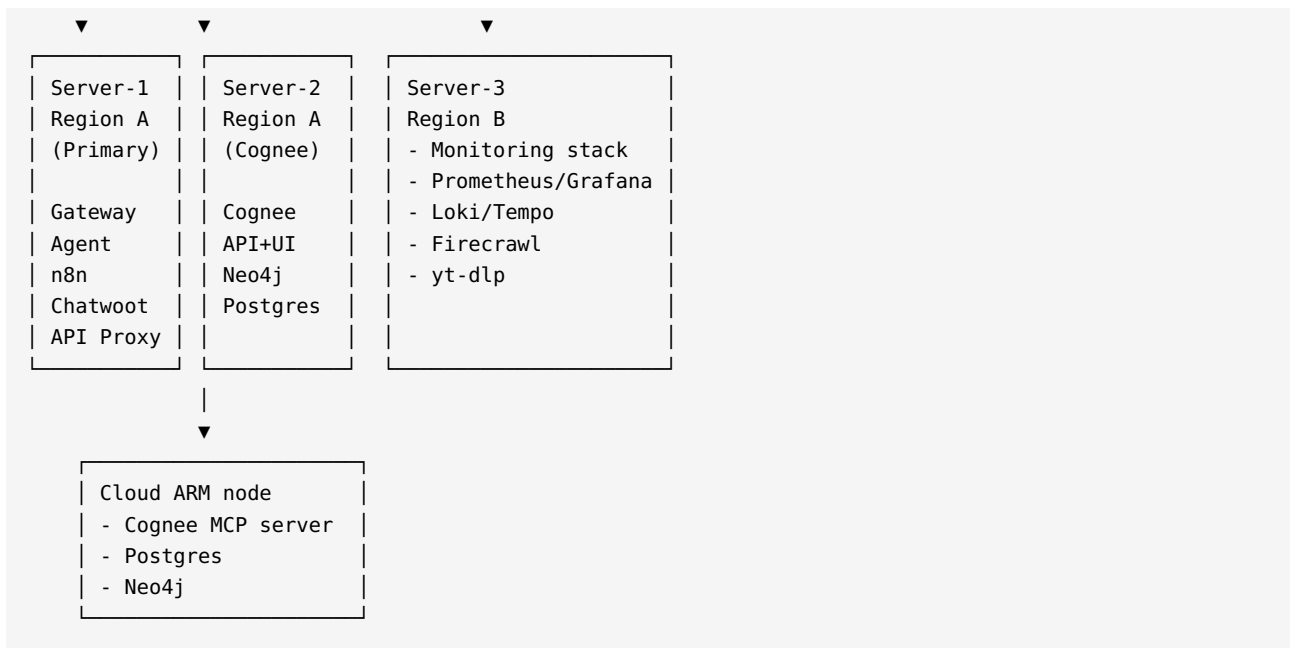
Chạy AI agent trong production về cơ bản khác với chạy web service truyền thống. Agent đưa ra quyết định, tiêu thụ API quota, spawn sub-process, ghi file, gọi dịch vụ bên ngoài, và nhìn chung hành động theo những cách khó dự đoán. Infrastructure hỗ trợ chúng cần mạnh mẽ, có thể quan sát được, và — quan trọng — an toàn trước chính các agent đó làm điều gì đó bất ngờ.

Chương này đề cập đến cách tôi xây dựng và vận hành infrastructure AI agent production trên nhiều server. Nó bao gồm những gì tôi làm đúng, những gì tôi làm sai thảm họa, và các config và pattern cụ thể hiện đang giữ mọi thứ hoạt động.

Topology Server

Trước khi đi vào các pattern triển khai, hãy hiểu bố cục vật lý. Hầu hết những gì tôi mô tả đều giả định thiết lập nhiều server này, mặc dù các pattern cũng áp dụng cho triển khai một server.





WireGuard mesh: VPN full mesh trên tất cả các node. Tất cả traffic liên server đều đi qua các tunnel được mã hóa (10.10.0.0/24). Đây là thứ làm cho VPN-binding cho các port nhạy cảm trở nên thực tế.

Phân công địa chỉ IP WireGuard:

- Server-3 (Region B): 10.10.0.1
- Server-1 (Region A): 10.10.0.2
- Server-2 (Region A): 10.10.0.3
- Load Balancer: 10.10.0.4
- Cloud ARM node: 10.10.0.5

Độ trễ từ Server-1: - Server-2: 0,9ms (cùng DC, P2P) - Cloud ARM: 2,0ms - Load Balancer: 3,5ms - Server-3 (Region B): 54ms (cross-region)

Kiểm Tra Thực Tế Process Manager: Docker vs Systemd vs PM2

Khi tôi bắt đầu chạy các dịch vụ AI agent, tôi mắc sai lầm chọn một process manager và cố dùng nó cho mọi thứ. Thực tế đã buộc tôi tiếp cận thực dụng hơn.

Kiểm Kế Dịch Vụ (Kiểm Tra Thực Tế Server-1)

Server-1 hiện chạy: - 14 Docker service - 8 Systemd service - 1 PM2 service (legacy, sắp được migrate)

Đó là 23 service được quản lý bởi 3 process manager khác nhau trên một server duy nhất. Có lý tưởng không? Không. Có thực tế không? Có. Đây là logic quyết định đằng sau mỗi lựa chọn.

Khi Nào Nên Dùng Docker

Docker là lựa chọn đúng khi: - Service có nhiều dependency (DB, Redis, message queue) - Bạn cần cô lập mạng giữa các thành phần - Service được thiết kế như một container (image chính thức tồn tại) - Bạn cần môi trường có thể tái tạo trên các server - Quản lý bí mật qua environment file là quan trọng

```
# docker-compose.yml pattern cho agent service
version: "3.8"

services:
  agent-gateway:
    image: agent-gateway:latest
    container_name: agent-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.agent-gateway.rule=Host(`api.youragent.example.com`)"
      - "traefik.http.routers.agent-gateway.tls.certresolver=letsencrypt"
      - "traefik.http.services.agent-gateway.loadbalancer.server.port=3000"

  redis:
    image: redis:7-alpine
    container_name: agent-redis
    restart: unless-stopped
    # QUAN TRỌNG: chỉ bind vào VPN interface để bảo mật
    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:
```

Khi Nào Nên Dùng Systemd

Systemd là lựa chọn đúng khi: - Service là một binary hoặc script đơn - Nó cần truy cập tài nguyên host (phần cứng, tính năng kernel cụ thể) - Thứ tự khởi động phụ thuộc vào host service quan trọng - Bạn cần giới hạn tài nguyên chi tiết qua cgroup - Service ra đời trước Docker hoặc việc container hóa nó thêm độ phức tạp không cần thiết

```
# /etc/systemd/system/agent-gateway.service
[Unit]
Description=AI Agent Gateway
After=network.target
Wants=network-online.target

[Service]
Type=simple
User=agent-user
Group=agent-user
WorkingDirectory=/home/agent-user/.agent-gateway
ExecStart=/usr/bin/node /home/agent-user/.agent-gateway/gateway/index.js
ExecReload=/bin/kill -HUP $MAINPID
Restart=on-failure
RestartSec=10
# Ngăn tiêu thụ tài nguyên không kiểm soát
MemoryMax=2G
CPUQuota=80%
# Environment
EnvironmentFile=/home/agent-user/.agent-gateway/.env
# Logging
StandardOutput=journal
StandardError=journal
SyslogIdentifier=agent-gateway

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

Tính năng systemd quan trọng cho agent service: - MemoryMax=2G — agent có thể tiêu thụ bộ nhớ không giới hạn nếu có bug trong quản lý context - CPUQuota=80% — ngăn agent chạy không kiểm soát làm đói các service khác - Restart=on-failure với RestartSec=10 — phục hồi tự động với backoff

Khi Nào Nên Dùng PM2 (Và Khi Nào Không)

PM2 phù hợp cho Node.js service nơi bạn muốn zero-downtime reload và cluster mode tích hợp. Tôi vẫn có một service trên PM2 (GitPocket API), phần lớn vì lý do lịch sử.

Đặc biệt đối với AI agent, PM2 có điểm yếu: giám sát process của nó không tích hợp tốt với monitoring stack dựa trên Docker. Nếu bạn đang sử dụng Prometheus + Grafana (điều bạn nên làm), systemd hoặc Docker cung cấp khả năng quan sát tốt hơn ngay từ đầu.

```
# PM2 config để tham khảo (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
    }
  ]
}
```

Khuyến nghị của tôi: Đừng giới thiệu PM2 cho agent service mới. Nếu bạn đang dùng nó, hãy migrate dần sang systemd hoặc Docker khi có cơ hội.

Traefik Với docker-socket-proxy (Cách Đúng)

Pattern phổ biến để chạy Traefik trong Docker là mount Docker socket trực tiếp:

```
# ĐỪNG LÀM THẾ NÀY
volumes:
  - /var/run/docker.sock:/var/run/docker.sock:ro
```

Vấn đề là bất kỳ ai có thể truy cập Traefik container thực chất có quyền root trên host. Vì AI agent có thể ảnh hưởng đến cấu hình container (chúng có quyền truy cập filesystem), đây là vector đe dọa thực sự.

Giải pháp là `docker-socket-proxy`, nằm giữa Traefik và Docker socket, chỉ expose các API mà Traefik thực sự cần.

```
# docker-compose.yml cho Traefik với 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:
  # Chỉ expose những gì Traefik cần
  CONTAINERS: 1
  SERVICES: 1
  NETWORKS: 1
  TASKS: 1
  # Từ chối rõ ràng các API nguy hiểm
  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@yourdomain.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 không có truy cập bên ngoài
  proxy-public:
    external: true

volumes:
  letsencrypt:

```

Cấu hình dynamic của Traefik cho service:

```

# /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 Cho Các Port Nhạy Cảm

Một trong những thực hành bảo mật quan trọng nhất cho thiết lập nhiều server là bind các service nhạy cảm vào VPN interface thay vì public IP. Điều này có nghĩa là service chỉ có thể tiếp cận từ trong mạng WireGuard.

Các service không bao giờ nên được công khai tiếp cận: - PostgreSQL (5432) - Redis (6379) - Prometheus (9090) - Grafana (3000, nếu không hướng đến công khai) - Internal agent API - Monitoring exporter

```

# PostgreSQL: chỉ bind vào VPN và localhost
# /etc/postgresql/14/main/postgresql.conf
listen_addresses = '127.0.0.1,10.10.0.2'

# Redis: cách tiếp cận tương tự
# /etc/redis/redis.conf
bind 127.0.0.1 10.10.0.2

# Prometheus: chỉ truy cập VPN
# Trong systemd unit hoặc docker command
--web.listen-address="10.10.0.2:9090"

```

Đối với Docker service, sử dụng port binding rõ ràng:

```

services:
  postgres:
    image: postgres:16
    ports:
      # Chỉ bind vào VPN IP, không phải 0.0.0.0
      - "10.10.0.2:5432:5432"
    environment:
      POSTGRES_PASSWORD: ${DB_PASSWORD}

```

Các quy tắc UFW để phù hợp:

```
# Cho phép truy cập DB chỉ từ 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

# Port công khai
ufw allow 80/tcp
ufw allow 443/tcp
ufw allow 51820/udp # WireGuard
```

HAProxy Cho Database Connection Pooling

AI agent có xu hướng kết nối nhiều với cơ sở dữ liệu. Một phiên agent duy nhất có thể mở hàng chục kết nối ngán. Nhân với nhiều instance agent và bạn nhanh chóng vượt quá giới hạn kết nối của PostgreSQL.

HAProxy giải quyết điều này với connection pooling ở cấp load balancer.

```
# /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 sang Server-2 nếu primary hỏng
    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
```

Với điều này, các agent service kết nối đến HAProxy frontend (10.10.0.4:5432), và HAProxy quản lý kết nối PostgreSQL thực sự một cách hiệu quả.

Câu Chuyện Chiến Trường #1: Bug Cô Lập Mạng Firecrawl

Ngày: Đầu tháng 2 năm 2026 **Mức độ:** Trung bình — service không thể truy cập từ server remote **Thời gian giải quyết:** ~30 phút khi đã chẩn đoán

Điều Đã Xảy Ra

Tôi triển khai Firecrawl trên Server-3 sử dụng Docker Compose chính thức. Mọi thứ hoạt động tốt khi tôi test từ Server-3. Tôi thiết lập routing Traefik, DNS trỏ đến `firecrawl.yourdomain.com`, và coi như xong.

Hai ngày sau, tôi cố gọi Firecrawl API từ Server-1 để crawl website cho một dự án knowledge base. Không có gì. Timeout liên tục.

```
# Từ Server-1:
curl https://firecrawl.yourdomain.com/v1/scrape
# Hoạt động (đi qua internet công khai → Traefik → Firecrawl)

# Từ Server-1 qua internal API call:
curl http://10.10.0.1:3002/v1/scrape
# Connection refused
```

Endpoint công khai hoạt động vì nó đi qua toàn bộ internet routing. Lờ gọi internal thất bại vì API service của Firecrawl chỉ lắng nghe trên localhost.

Nguyên Nhân Gốc

File `docker-compose.yml` của Firecrawl có điều này trong cấu hình API service:

```
# Cấu hình Firecrawl mặc định (vấn đề)
services:
  api:
    ports:
      - "127.0.0.1:3002:3002"
```

Binding `127.0.0.1:3002:3002` có nghĩa: expose port 3002 trên host, nhưng chỉ lắng nghe trên loopback interface. Bất kỳ kết nối nào từ IP bên ngoài (bao gồm cả VPN tunnel traffic) bị từ chối ngay lập tức.

Cách Sửa

```
# Cấu hình đã sửa
services:
  api:
    ports:
      # 0.0.0.0 có nghĩa "tất cả interface" bao gồm VPN
      - "0.0.0.0:3002:3002"
      # Hoặc dạng ngắn hơn:
      # - "3002:3002"
```

Sau đó cập nhật UFW để cho phép truy cập trên VPN interface:

```
# Cho phép Firecrawl API chỉ từ mạng VPN
ufw allow in on wg0 to any port 3002
# Chặn từ internet công khai (Traefik xử lý truy cập công khai)
ufw deny 3002/tcp
```

Bài Học

Khi một service cần có thể gọi được từ các server khác trong mạng của bạn, Docker port binding `127.0.0.1:PORT:PORT` sẽ thầm lặng chặn bạn. Triệu chứng — `connection refused` thay vì lỗi mạng — có thể khiến bạn nghĩ service đang hỏng trong khi thực ra nó chỉ không thể tiếp cận.

Kiểm tra port binding của bạn với:

```
docker ps --format "table {{.Names}}\t{{.Ports}}"
# hoặc chi tiết hơn:
ss -tlnp | grep docker
```

Bài học mở rộng ra ngoài Firecrawl. Bất kỳ service nào bạn triển khai cần có khả năng truy cập cross-server nên sử dụng binding `0.0.0.0` (được bảo mật ở cấp firewall với quy tắc UFW) thay vì binding `127.0.0.1`.

Câu Chuyện Chiến Trường #2: Server Không Có Firewall

Ngày: Tháng 2 năm 2026, trong quá trình kiểm toán bảo mật **Mức độ:** Cao — server có thể truy cập công khai không có firewall trong thời gian không xác định **Thời gian giải quyết:** ~2 giờ (hardening cẩn thận để tránh khóa chính mình ra ngoài)

Điều Đã Xảy Ra

Tôi đang thực hiện kiểm toán bảo mật định kỳ trên tất cả server. Checklist bao gồm xác minh UFW đang hoạt động và được cấu hình đúng trên mỗi server. Server-1 và Server-2 ổn. Rồi tôi đến Server-3:

```
# Trên Server-3:  
sudo ufw status  
# Status: inactive
```

Không phải “active nhưng thiếu một số quy tắc.” Không phải “được bật với policy cho phép.” Chỉ là: **inactive**. Firewall chưa bao giờ được bật. Server-3, lưu trữ monitoring stack (Prometheus, Grafana, Loki, Tempo), đã chạy với tất cả port mở rộng rãi cho internet trong toàn bộ thời gian hoạt động.

Đánh Giá Thiệt Hại

```
# Kiểm tra những gì thực sự có thể truy cập  
ss -tlnp  
  
# Phát hiện:  
# 0.0.0.0:9090 → Prometheus (bất kỳ ai cũng có thể query metric của bạn)  
# 0.0.0.0:3000 → Grafana (bất kỳ ai cũng có thể đọc dashboard)  
# 0.0.0.0:3100 → Loki (bất kỳ ai cũng có thể query log)  
# 0.0.0.0:9411 → Tempo traces  
# 0.0.0.0:9100 → node_exporter (metric hệ thống đầy đủ)
```

Prometheus metric không “nguy hiểm” theo nghĩa truyền thống — chúng không cho bạn truy cập hệ thống trực tiếp. Nhưng chúng tiết lộ nhiều thông tin vận hành: bạn đang chạy service gì, sức khỏe của chúng, pattern tiêu thụ tài nguyên, thậm chí tên endpoint nội bộ nếu bạn đang scraping service discovery.

Grafana không có authentication là vấn đề nghiêm trọng hơn. Bất kỳ ai tìm thấy endpoint đều có thể đọc dashboard của bạn, và nếu thông tin xác thực admin là mặc định hoặc yếu, họ có thể sửa đổi chúng.

Quy Trình Hardening An Toàn

Phần nguy hiểm của việc bật UFW trên server đang hoạt động là vô tình chặn SSH và khóa chính mình ra ngoài. Tôi đã thấy điều này xảy ra. Không vui.

```
# QUAN TRỌNG: Thực hiện theo đúng thứ tự này  
  
# Bước 1: Cho phép SSH TRƯỚC KHI bật UFW  
sudo ufw allow OpenSSH  
sudo ufw allow 22/tcp # Phòng hờ
```

```

# Bước 2: Cho phép WireGuard
sudo ufw allow 51820/udp

# Bước 3: Cho phép service công khai
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp

# Bước 4: Cho phép monitoring chỉ từ VPN
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

# Bước 5: Từ chối mọi thứ khác (ngầm trong UFW, nhưng rõ ràng là tốt hơn)
sudo ufw default deny incoming
sudo ufw default allow outgoing

# Bước 6: Bật - BÂY GIỜ (không phải trước đây)
sudo ufw enable

# Bước 7: Xác minh ngay lập tức
sudo ufw status verbose

```

Nếu bạn làm điều này từ xa qua SSH, vẫn có rủi ro. Test SSH rule của bạn với một kết nối mới (trong terminal riêng biệt) trước khi đóng phiên hiện có.

```

# Trong terminal 2, trước khi đóng terminal 1:
ssh user@server-3 "echo 'SSH still works'"

```

Chỉ sau khi xác nhận SSH vẫn hoạt động thì đóng terminal 1.

Những Gì Chúng Tôi Thêm Vào Checklist Kiểm Toán

Sau sự cố này, mỗi lần onboard server đều bao gồm:

```

#!/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"
```

Chạy lệnh này trên mỗi server mới trước khi kết nối nó vào mạng của bạn.

Pattern Docker Compose Cho Agent Service

Pattern 1: Agent Với Mạng Cô Lập

```
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 # Chỉ nếu cần truy cập bên ngoài
    volumes:
      - agent-data:/data
      - /home/agent-user/.agent-gateway/workspace:/workspace:ro # Workspace chỉ đọc
    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"
    # Giới hạn tài nguyên – quan trọng cho agent
    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 # Không có trên 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

Cho các thiết lập agent phức tạp hơn với nhiều thành phần:

```
version: "3.8"  
  
x-common: &common  
  restart: unless-stopped  
  logging:  
    driver: "json-file"  
    options:  
      max-size: "100m"  
      max-file: "3"  
  
services:  
  gateway:  
    <<: *common  
    image: agent-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: agent-worker:latest  
    depends_on:  
      - gateway  
    scale: 2 # Chạy 2 worker instance  
  
  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" # Chỉ VPN  
  
  redis:  
    <<: *common  
    image: redis:7-alpine  
    command: redis-server --requirepass ${REDIS_PASSWORD}
```

```
ports:
  - "10.10.0.2:6379:6379" # Chỉ VPN

volumes:
  postgres-data:
```

Pattern 3: Systemd Unit Cho Agent Process

Khi chạy agent như system service (không trong Docker):

```
# /etc/systemd/system/agent-worker@.service
# Template unit – tạo instance với: systemctl start agent-worker@1

[Unit]
Description=AI Agent Worker Instance %i
After=network-online.target
Requires=network-online.target

[Service]
Type=simple
User=agent-user
Group=agent-user
WorkingDirectory=/home/agent-user/agent

# @ làm cho đây là template – %i là số instance
ExecStart=/home/agent-user/.venv/bin/python -m agent.worker --instance %i

# Restart policy
Restart=on-failure
RestartSec=15
StartLimitBurst=3
StartLimitIntervalSec=120

# Giới hạn tài nguyên
MemoryMax=1G
CPUQuota=50%
TasksMax=64

# Environment
EnvironmentFile=/home/agent-user/agent/.env
Environment="WORKER_ID=%i"
Environment="LOG_LEVEL=info"

# File descriptor limit (agent có thể mở nhiều kết nối)
LimitNOFILE=65536

# Logging
StandardOutput=journal
StandardError=journal
SyslogIdentifier=agent-worker-%i

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

Khởi động nhiều worker:

```
systemctl enable --now agent-worker@1
systemctl enable --now agent-worker@2
systemctl enable --now agent-worker@3
```

Quản Lý Bí Mật

Không bao giờ hardcoded bí mật trong docker-compose file hoặc systemd unit. Sử dụng environment file được loại trừ khỏi version control.

```
# .env file (không bao giờ commit cái này)
ANTHROPIC_API_KEY=sk-ant-...
OPENAI_API_KEY=sk-...
REDIS_PASSWORD=...
POSTGRES_PASSWORD=...
TELEGRAM_BOT_TOKEN=...

# .gitignore
.env
*.env
.env.*
!.env.example
```

Cho thiết lập mạnh mẽ hơn, sử dụng Vault:

```
# Lấy bí mật từ Vault khi khởi động service
#!/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)

# Lấy bí mật và export
eval $(vault kv get -format=json secret/agent | \
jq -r '.data.data | to_entries | .[] | "export \(.key)=\(.value)"')

exec "$@"
```

Sau đó trong systemd unit:

```
ExecStart=/usr/local/bin/start-with-vault.sh /home/agent-user/.venv/bin/python -m agent
```

Health Check Và Quản Lý Phụ Thuộc

Agent service AI thường có dependency khởi động phức tạp. Đừng chỉ dùng `depends_on` với tên service — dùng `condition: service_healthy` với health check phù hợp.

```

services:
  agent:
    depends_on:
      database:
        condition: service_healthy
      redis:
        condition: service_healthy
      model-api:
        condition: service_started # Service bên ngoài, không thể 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

```

Không có điều này, agent service của bạn có thể khởi động trước khi cơ sở dữ liệu sẵn sàng, không kết nối được, và sau đó không retry đúng cách. Container xuất hiện “running” nhưng agent thực sự bị hỏng.

Checklist Triển Khai

Trước khi triển khai bất kỳ agent service mới nào lên production:

Trước khi triển khai:

- [] Port binding sử dụng interface rõ ràng (không phải 0.0.0.0 cho service nội bộ)
- [] Quy tắc UFW đã thêm cho các port mới
- [] Giới hạn tài nguyên được đặt (memory, CPU)
- [] .env file có mật và bị loại trừ khỏi git
- [] Health check được cấu hình
- [] Restart policy được đặt là unless-stopped hoặc on-failure

Sau khi triển khai:

- [] Xác minh service đang chạy: docker ps hoặc systemctl status
- [] Test từ server khác (không chỉ localhost)
- [] Kiểm tra log để phát hiện lỗi khởi động: docker logs hoặc journalctl
- [] Xác minh routing Traefik nếu public-facing
- [] Thêm vào monitoring (xem Chương 9)
- [] Ghi lại trong service inventory

Bảo mật:

- [] Service chạy với non-root user
- [] Port nhạy cảm không được expose ra internet công khai
- [] Bí mật được lấy từ env file, không phải hardcoded
- [] Docker socket không được mount trừ khi thực sự cần thiết

Tóm Tắt

Triển khai AI agent lên production đòi hỏi nhiều sự cẩn thận hơn so với triển khai web service truyền thống. Agent đưa ra quyết định tự động, có nghĩa là các lỗi infrastructure bị bắt gặp nhanh trong triển khai truyền thống có thể không được chú ý cho đến khi chúng gây ra thiệt hại đáng kể.

Các nguyên tắc chính từ chương này:

1. **Sử dụng đúng process manager** — Docker cho service có dependency, systemd cho service cấp hệ thống, và tránh PM2 cho các triển khai mới
2. **VPN-bind các port nhạy cảm** — không bao giờ expose cơ sở dữ liệu hoặc monitoring ra internet công khai
3. **Sử dụng docker-socket-proxy** — dùng cho Traefik (hoặc bất kỳ service nào) truy cập trực tiếp vào Docker socket
4. **Kiểm tra port binding** — `127.0.0.1:PORT:PORT` vs `0.0.0.0:PORT:PORT` không rõ ràng nhưng quan trọng cho thiết lập nhiều server
5. **Bật firewall của bạn** — điều này nghe có vẻ hiển nhiên cho đến khi bạn tìm thấy server đã chạy không có firewall trong nhiều tháng
6. **Đặt giới hạn tài nguyên** — agent có thể tiêu thụ tài nguyên không giới hạn; giới hạn ngăn một agent kém làm sập toàn bộ server của bạn

Chương tiếp theo đề cập đến quản lý chi phí, vì tất cả infrastructure này cần chạy AI model mà tốn tiền thật.

Chương 8: Quản Lý Chi Phí — Đùng Phá Sản Khi Chạy AI Agent

“Tôi nghĩ đó chỉ là một thí nghiệm nhỏ. Tôi chỉ để nó chạy qua đêm. Sáng hôm sau tôi mở Google Cloud Console và cảm thấy dạ dày mình thắt lại.”

Chi phí infrastructure AI có một thuộc tính đặc biệt mà chi phí cloud truyền thống không có: chúng có thể tăng theo hàm mũ trong vài giờ, không phải vài ngày. Một agent chạy bất kiểm soát cứ retry mãi một task thất bại, hoặc một process bị bỏ quên liên tục poll API mỗi vài giây, có thể tích lũy hàng trăm đô la trước khi ai đó nhận ra. Không giống như một EC2 instance tính phí theo giờ có thể dự đoán, chi phí API LLM gắn liền với lượng token tiêu thụ — và lượng token tiêu thụ có thể hoàn toàn không thể đoán trước.

Chương này nói về chi phí thực sự của việc chạy AI agent trong production: hóa đơn \$500 mà tôi không ngờ, mức giảm 91% token tôi cuối cùng đạt được, và các hệ thống tôi xây dựng để đảm bảo nó không bao giờ xảy ra nữa.

Sự Cố Gemini \$500

Ngày: Cuối năm 2025 **Hóa đơn:** ~\$500 USD trong khoảng 48 giờ **Hoàn tiền:** Một lần hoàn tiền lịch sự từ Google

Điều Đã Xảy Ra

Tôi đang thử nghiệm một batch processing pipeline — sử dụng Gemini Pro để phân tích và phân loại một dataset lớn. Job được khởi động thủ công, chạy một lúc, rồi tôi bị phân tâm với công việc khác. Điều tôi không nhận ra là job đã gặp lỗi giữa chừng và bước vào vòng lặp retry.

Logic retry thật đơn giản: khi gặp bất kỳ lỗi nào, sleep 5 giây và thử lại. Lỗi trong trường hợp này là một vấn đề API thoáng qua cứ lặp đi lặp lại. Pipeline tiếp tục thực hiện lời gọi API, tạo ra output token, và tính phí tài khoản của tôi theo giá Gemini Pro.

Khi tôi nhận ra (kiểm tra Google Cloud Console sáng hôm sau), job đã chạy được khoảng 48 giờ và tích lũy gần \$500 phí.

```

# Pattern retry có vấn đề (ĐỪNG làm thế này)
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 # Cứ tiếp tục mãi

```

Điều Lẽ Ra Phải Xảy Ra

```

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 với bảo vệ ngân sách.
    Raise sau max_retries hoặc nếu ngân sách cạn kiệt.
    """
    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 # Đừng nuốt lỗi cuối cùng
            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")

```

Cách Lấy Lại Tiền

Tôi liên hệ Google Cloud Support và giải thích tình huống thành thật: một process chạy bất kiểm soát, tôi không nhận ra kịp thời, đây là biểu đồ sử dụng cho thấy đột tăng đột biến. Google đã hoàn tiền một lần lịch sử.

Từ khóa là **một lần**. Họ nói rõ điều đó. Nếu xảy ra lại, bạn tự lo.

Tôi hỏi nhân viên hỗ trợ họ khuyến nghị gì để ngăn điều này. Câu trả lời: cảnh báo ngân sách. Cụ thể, đặt chúng đủ thấp để bạn được thông báo trong khi vẫn còn thời gian để phản ứng.

Cảnh Báo Ngân Sách Không Phải Tùy Chọn

Mỗi AI API bạn sử dụng phải có cảnh báo ngân sách được cấu hình. Đây không phải thứ tốt để có. Đây là thứ đầu tiên bạn thiết lập, trước khi viết một dòng code.

Cảnh Báo Ngân Sách Google Cloud

```
# Tạo ngân sách qua 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
```

Nhưng cảnh báo không thôi chưa đủ — bạn cần hành động dựa trên chúng. Thiết lập Pub/Sub subscriber thực sự làm gì đó:

```
# 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 # Giết process tồn kém ở mức 90% ngân sách

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')}")

)

# Gửi thông báo Telegram
requests.post(
    f"https://api.telegram.org/bot{TELEGRAM_BOT_TOKEN}/sendMessage",
    json={"chat_id": TELEGRAM_CHAT_ID, "text": alert_text}
)

# Ở mức 90%, giết các batch job không quan trọng
if percent_used >= ALERT_THRESHOLD_KILL:
    kill_batch_jobs()

def kill_batch_jobs():
    """Dừng bất kỳ batch processing job nào đang chạy."""
    import subprocess
    # Giết bất kỳ process nào được gắn tag là batch job
    subprocess.run(["pkill", "-f", "batch_processor"], check=False)
    # Gửi cảnh báo bổ sung
    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"
        }
    )
)

```

Ngân Sách Anthropic Claude

Anthropic không có API ngân sách thời gian thực giống Google Cloud, nhưng bạn có thể đặt giới hạn chi tiêu trong console và sử dụng rate limiting dựa trên sử dụng ở phía bạn:

```

# cost_tracker.py - Theo dõi chi tiêu Claude API theo thời gian thực
import anthropic
from datetime import datetime
import json
import os

DAILY_BUDGET_USD = 10.0
COST_FILE = "/tmp/claude_daily_cost.json"

# Chi phí xấp xỉ mỗi 1M token (Claude Sonnet 4.6 đầu năm 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
    )

    # Theo dõi chi phí
    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

```

Tín Dụng Dùng Thử Và Các Model Không Áp Dụng

Một phát hiện đau đớn: **tín dụng dùng thử không áp dụng cho tất cả model**. Khi tôi đăng ký Google AI Studio lần đầu, tôi có \$300 tín dụng dùng thử. Tôi cho rằng điều này bao gồm mọi thứ. Không phải vậy.

Một số model Gemini — cụ thể là những model hữu ích nhất cho các task agent phức tạp — được tính phí riêng ngay cả trong thời gian dùng thử. Tôi phát hiện điều này khi nhận được hóa đơn nhỏ (\$12) trong khoảng thời gian tôi cho là đang dùng thử miễn phí.

Giá Google AI (xấp xỉ, hãy xác minh giá hiện tại):

Model	Input (mỗi 1M)	Output (mỗi 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

Giá Anthropic Claude (xấp xỉ):

Model	Input (mỗi 1M)	Output (mỗi 1M)
Claude Haiku 4	\$0,25	\$1,25
Claude Sonnet 4.6	\$3,00	\$15,00
Claude Opus 4	\$15,00	\$75,00

Quy tắc: Luôn test với model rẻ nhất hoạt động được. Chỉ nâng cấp lên model đắt hơn khi bạn có yêu cầu năng lực cụ thể mà model rẻ hơn không đáp ứng được.

Khủng Hoảng Tối Ưu Token

Vào tháng 2 năm 2026, hơn một chục tài khoản provider của tôi bị tạm đình chỉ đồng thời. Điều này cắt đứt quyền truy cập API miễn phí của tôi từ hàng trăm request/tuần xuống gần như bằng không qua một đêm. Nó buộc phải thực hiện sprint tối ưu token khẩn cấp.

Vấn Đề: Lượng Token Cơ Sở

Trước khi tối ưu, đây là những gì được inject vào mỗi context agent:

File context được inject mỗi request:

File	Token	Nội dung
AGENTS.md	~3.535	Danh sách agent đầy đủ + ví dụ
MEMORY.md	~2.833	Bộ nhớ vận hành đầy đủ
TOOLS.md	~838	Tài liệu tool hoàn chỉnh
SOUL.md	~636	Hướng dẫn tính cách mở rộng
HEARTBEAT.md	~569	Logic heartbeat đầy đủ
BOOTSTRAP.md	~360	Quy trình khởi động
USER.md	~158	Tùy chọn người dùng
IDENTITY.md	~101	Cấu hình nhận dạng
TỔNG CỘNG	~9.030	

9.030 token mỗi request, chỉ cho system context. Với hàng chục request mỗi giờ, con số cộng dồn rất nhanh.

Tối Ưu Hóa

Hiểu biết là hầu hết context này là thông tin tham chiếu — chi tiết bạn chỉ cần khi đang làm việc tích cực trên một task cụ thể, không phải trong mỗi lần gọi duy nhất.

Sau khi tối ưu:

File	Trước	Sau	Giảm
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%
TỔNG CỘNG	~9.030	~797	-91%

Giảm 91%. Từ ~9.000 token xuống ~800 token mỗi lần tải context.

Làm thế nào? Nguyên tắc chính là: **đừng inject dữ liệu tham chiếu, inject con trỏ đến dữ liệu tham chiếu.**

Thay vì:

```
# AGENTS.md (3.535 token)
## Available Agents

### researcher
The researcher agent conducts deep technical research...
[hàng trăm dòng tài liệu]

### planner
The planner agent creates implementation plans...
[hàng trăm dòng nữa]

### code-reviewer
...
```

Sau:

```
# AGENTS.md (270 token)
Agents: researcher, planner, code-reviewer, tester, debugger, docs-manager
```

Full docs: docs/agents/
Use: spawn `<agent-name>` for specialized tasks

Agent vẫn có thể truy cập tài liệu đầy đủ khi cần — nó chỉ không tải tất cả trong mỗi lần gọi duy nhất.

Tối Ưu Khoảng Cách Heartbeat

Orchestrating agent có routine heartbeat chạy mỗi 3 phút. Mỗi heartbeat kích hoạt context load, memory sync, và health check. Đó là 20 lời gọi API mỗi giờ chỉ cho heartbeat, đốt cháy token ngay cả khi agent đang nhàn rỗi.

Cách sửa: tăng khoảng cách và tách heartbeat nhẹ khỏi thao tác nặng.

```
Trước:
- Heartbeat mỗi 3 phút
- Mỗi heartbeat: tải context đầy đủ + chạy health check + sync memory
- Chi phí: ~20 lời gọi API nặng/giờ khi nhàn rỗi

Sau:
- Heartbeat mỗi 30 phút
- Mỗi heartbeat: chỉ kiểm tra trạng thái tối giản
- Health check: chuyển sang systemd timer (không token AI)
- Memory sync: kích hoạt theo sự kiện, không theo lịch
- Chi phí: ~2 lời gọi API nhẹ/giờ khi nhàn rỗi
```

Heartbeat giảm từ đốt cháy 20+ lời gọi API/giờ xuống còn 2. Giảm 10 lần chi phí nhàn rỗi.

Xóa Tất Cả Cron Job AI Tự Động

Quyết định triệt để nhất là xóa tất cả cron job AI tự động. Tất cả mọi thứ.

Đây là lý do: mỗi cron job health check đang chạy AI model để làm những việc không cần intelligence. Kiểm tra xem container có đang chạy không là vấn đề đã được giải quyết — `docker inspect` có thể cho bạn biết trong mili giây. Gửi cảnh báo Telegram là một lời gọi HTTP đơn giản. Không có gì trong số đó cần AI model.

```
# Trước: Agent cron job chạy mỗi vài phút
# Chúng đều tiêu thụ AI token:
# - auto-heal-alerts (mỗi 1 phút)
# - memory-auto-save (mỗi 5 phút)
# - devops-health-check (mỗi 6 giờ)

# Sau: Systemd timer không tốn token AI
```

```

# /etc/systemd/system/health-check.service
[Unit]
Description=Server Health Check (no AI)
After=network.target

[Service]
Type=oneshot
User=agent-user
ExecStart=/home/agent-user/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/agent-user/scripts/health-check.sh
# Pure bash health check – không tốn token AI

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
}

# Kiểm tra các container quan trọng
CRITICAL_CONTAINERS=("agent-gateway" "traefik" "postgres" "redis")

for container in "${CRITICAL_CONTAINERS[@]}; do
    STATUS=$(docker inspect --format '{{.State.Status}}' "$container" 2>/dev/null)
    if [ "$STATUS" != "running" ]; then
        # Cố gắng 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

```

Thời điểm duy nhất bạn nên sử dụng AI trong health check là khi bạn cần năng lực AI — chẩn đoán vấn đề phức tạp, tạo ra kế hoạch khắc phục, thứ gì đó thực sự cần intelligence. Kiểm tra xem port có mở không không đủ điều kiện.

Orphan tmux Session: Kẻ Giết Quota Thâm Lặng

Cái này mất một lúc để tôi chẩn đoán. Tôi tiếp tục nhận thấy lượng quota tiêu thụ mà tôi không thể giải thích. Kiểm tra các phiên đang hoạt động không cho thấy gì rõ ràng. Rồi tôi nhìn vào 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)
```

Bốn tmux session, một số đã cũ nhiều ngày. Bên trong mỗi session, một process Claude Code hoặc gateway agent vẫn đang chạy, chờ input, định kỳ làm mới context, tiêu thụ token trong vòng lặp heartbeat.

Session old-experiment từ ngày 2 tháng 2 đã thâm lặng đốt cháy quota trong bốn ngày.

```
# Tìm và kiểm toán tất cả tmux session
tmux ls -F "#{session_name}: created #{session_created_string}"

# Xem bên trong từng session
tmux capture-pane -p -t claude-code

# Dọn dẹp các session cũ không nên chạy
tmux kill-session -t old-experiment
tmux kill-session -t debug-session
```

Bây giờ tôi chạy kiểm toán hàng tuần:

```
#!/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 "=== Gateway agent processes ==="
```

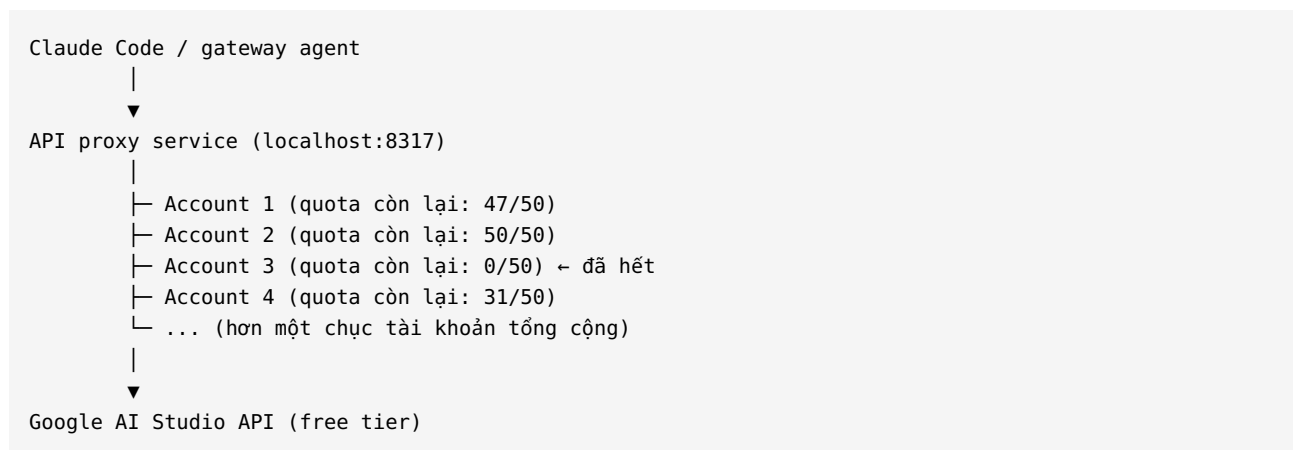
```
pgrep -la "agent-gateway\|gateway" 2>/dev/null || echo "No gateway processes"

echo ""
echo "=== Orphan Python AI scripts ==="
pgrep -la "python" | grep -E "batch|process|generate|crawl" || echo "None found"
```

API Proxy Cục Bộ: Quota Miễn Phí Qua Account Rotation

Sau khi mất các tài khoản provider, tôi xây dựng lại chiến lược API miễn phí bền vững hơn sử dụng API proxy cục bộ — một service round-robin qua nhiều tài khoản Google AI Pro.

Cách Hoạt Động



Proxy sử dụng chiến lược `fill-first`: làm cạn kiệt hoàn toàn một tài khoản trước khi chuyển sang tài khoản tiếp theo. Điều này tối đa hóa việc sử dụng quota mỗi tài khoản vì free tier của Google được reset hàng tuần.

```
# Kiểm tra tài khoản nào đang hoạt động
journalctl -u api-proxy --no-pager -n 30 | grep "Use OAuth"
# Output: Use OAuth provider=google auth_file=account3@gmail.com.json

# Xem phân phối request trên các tài khoản
journalctl -u api-proxy --no-pager --since "1 week ago" | \
  grep "Use OAuth" | \
  awk '{print $NF}' | \
  sort | uniq -c | sort -rn
```

Thực Tế Quota

Quota Google AI Pro: 50 request/tuần mỗi tài khoản.

Không phải 50 mỗi ngày. Mỗi tuần. Điều này làm tôi bất ngờ — tôi kỳ vọng ít nhất reset hàng ngày. Toán học thực tế:

15 tài khoản × 50 request/tuần = 750 request/tuần tổng cộng
750 / 7 ngày = ~107 request/ngày tối đa

Với các task agent phức tạp trung bình 5-10 request mỗi task:
= 10-21 task agent có ý nghĩa mỗi ngày (free tier)

Điều này đủ cho phát triển và sử dụng production vừa phải, nhưng một AI agent bận rộn trong production sẽ hết nhanh. Lên kế hoạch phù hợp.

Đánh Đổi Độ Trễ

Có hình phạt độ trễ thực sự với proxy cục bộ so với lời gọi API trực tiếp:

So Sánh Độ Trễ:

Phương pháp	Độ trễ	Chi phí
Gemini Flash API trực tiếp	680-914ms	\$0,075/1M token
Proxy cục bộ (Google AI Pro)	2000-2300ms	Miễn phí

Proxy cục bộ chậm hơn 2-3 lần vì OAuth overhead và proxy routing. Đối với task agent tương tác, điều này rõ ràng. Đối với batch processing không quan tâm đến độ trễ, đây là đánh đổi hợp lý để có quyền truy cập miễn phí.

Chiến Lược Phân Cấp Model

Không phải task nào cũng cần model có năng lực nhất (và đắt nhất). Chiến lược phân cấp route task đến model rẻ nhất có thể xử lý chúng.

Ma Trận Quyết Định Phân Cấp Model:

LOẠI TASK	MODEL	LÝ DO
Lý luận, lập kế hoạch phức tạp	Claude Opus 4	Chất lượng quan trọng
Tạo code (phức tạp)	Claude Sonnet	Cân bằng
Tạo code (đơn giản)	Claude Haiku	Tốc độ + chi phí
Phân loại text	Gemini Flash	Rẻ + nhanh
Tóm tắt	Gemini Flash	Khối lượng cao
Tạo embedding	text-embedding	Batch-optimized
Phân tích ảnh	Gemini Pro	Cần vision
Q&A / routing đơn giản	Claude Haiku	Rẻ nhất có năng lực

Triển khai trong code:

```
from enum import Enum
from typing import Literal
```

```

class TaskComplexity(Enum):
    SIMPLE = "simple"      # Phân loại, routing, có/không
    MODERATE = "moderate" # Tạo code, tóm tắt
    COMPLEX = "complex"   # Lý luận, lập kế hoạch, phân tích
    CRITICAL = "critical" # Quyết định production, bảo mật

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 đến model phù hợp rẻ nhất."""
    return MODEL_TIERS[complexity]

# Cách sử dụng
def analyze_log_entry(log_line: str) -> dict:
    # Phân loại log đơn giản – dùng Haiku
    model = select_model(TaskComplexity.SIMPLE)
    # ...

def generate_deployment_plan(requirements: str) -> str:
    # Lập kế hoạch triển khai phức tạp – dùng Opus
    model = select_model(TaskComplexity.COMPLEX)
    # ...

```

Batch API: Giảm 50% Chi Phí

Cả Claude và Gemini đều cung cấp batch processing API với giảm giá đáng kể. Nếu bạn có công việc không cần response thời gian thực — tạo mô tả cho 1000 sản phẩm, phân tích log tuần trước, tạo test case — batch API có thể cắt giảm chi phí xuống một nửa.

```

# Ví dụ Anthropic Batch API
import anthropic

client = anthropic.Anthropic()

# Gửi batch request
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}")
# Kết quả có trong vòng 24h, với 50% chi phí bình thường

# Google Gemini Batch API
import google.generativeai as genai

# Tạo 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/"}
    }
)
# Giảm 50% so với API đồng bộ

```

Dashboard Giám Sát Chi Phí

Đây là thiết lập Prometheus + Grafana để theo dõi chi phí API AI theo thời gian thực:

```

# cost_metrics_exporter.py
from prometheus_client import start_http_server, Counter, Gauge, Histogram
import time

# Định nghĩa metric
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):
    """Gọi hàm này sau mỗi API request để theo dõi metric."""
    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)

```

Cấu hình scrape Prometheus:

```

# prometheus.yml (thêm vào scrape_configs)
scrape_configs:
  - job_name: 'ai_cost_metrics'
    static_configs:
      - targets: ['10.10.0.2:8091']
    scrape_interval: 60s

```

Grafana dashboard (các panel chính):

```

Panel 1: Chi tiêu hàng ngày theo provider (bar chart, 30 ngày qua)
query: increase(ai_api_cost_usd_total[1d])

Panel 2: Tốc độ tiêu thụ token (time series)
query: rate(ai_api_tokens_total[5m]) * 300

Panel 3: Ngân sách còn lại (gauge)
query: ai_daily_budget_remaining_usd

Panel 4: Chi phí theo model (pie chart)
query: sum by (model) (ai_api_cost_usd_total)

Panel 5: Độ trễ P95 theo provider (time series)
query: histogram_quantile(0.95, rate(ai_api_request_duration_seconds_bucket[5m]))

```

Chi Phí Của “Chỉ Kiểm Tra”

Một pattern tôi thấy liên tục — bao gồm trong thiết lập của chính tôi trước khi tôi bắt được — là agent sử dụng AI cho các task không cần AI.

```

# Anti-pattern phổ biến: dùng AI cho kiểm tra đơn giản
async def is_server_healthy(server_ip: str) -> bool:
    response = await call_claude(
        f"Check if server {server_ip} is healthy and responding"
    )
    # Điều này tốn token cho thứ mà ping/curl có thể làm ngay lập tức

```

```

# Tốt hơn: Chỉ dùng AI khi thực sự cần intelligence
import subprocess
import httpx

def is_server_healthy(server_ip: str, port: int = 80) -> bool:
    """Kiểm tra sức khỏe không tốn token AI."""
    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:
    """Chỉ dùng AI khi bạn có chẩn đoán phức tạp cần thực hiện."""
    return await call_claude(
        f"Server {server_ip} is down with symptoms: {symptoms}. "
        f"What are the most likely causes and remediation steps?"
    )

```

Quy tắc: nếu bạn có thể làm bằng bash, hãy làm bằng bash.

Script Giám Sát Ngân Sách

Một script thực tế để review chi phí hàng ngày:

```

#!/bin/bash
# /home/agent-user/scripts/daily-cost-report.sh
# Chạy qua cron: 0 8 * * * /home/agent-user/scripts/daily-cost-report.sh

TELEGRAM_BOT_TOKEN="${TELEGRAM_BOT_TOKEN}"
TELEGRAM_CHAT_ID="${TELEGRAM_CHAT_ID}"

# Thu thập metric từ 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]['value'][1] if d['data']
    ['result'] else '0')"
}

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])")

# Lấy trạng thái tài khoản API proxy
ACTIVE_ACCOUNT=$(journalctl -u api-proxy --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}
Proxy 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"
```

Các Quy Tắc Quản Lý Chi Phí Chính

Sau nhiều tháng chạy AI agent trong production và trả tiền cho những sai lầm như sự cố Gemini \$500, đây là các quy tắc tôi tuân theo mà không có ngoại lệ:

1. **Đặt cảnh báo ngân sách trên mọi tài khoản API, ngay lập tức, trước khi viết một dòng code**
2. **Sử dụng exponential backoff với giới hạn retry tối đa** — không bao giờ retry vô hạn
3. **Route task đến model rẻ nhất có thể xử lý chúng** — dùng Haiku cho công việc đơn giản, Opus chỉ khi cần thiết
4. **Chạy health check infrastructure bằng bash, không phải AI** — mỗi AI health check là một lần đốt cháy token
5. **Kiểm toán tmux session hàng tuần** — orphan session thâm lặng tiêu thụ quota
6. **Sử dụng batch API cho công việc không cần thời gian thực** — giảm 50% là đáng kể ở quy mô
7. **Theo dõi lượng token tiêu thụ trong Prometheus** — bạn không thể quản lý những gì bạn không đo lường
8. **Đọc điều khoản nhỏ về tín dụng dùng thử** — chúng thường không áp dụng cho model premium
9. **Giữ file context gọn nhẹ** — inject con trỏ, không phải tài liệu đầy đủ
10. **Khoảng cách heartbeat nên dài** — 30 phút là ổn cho hầu hết heartbeat agent; 3 phút là lãng phí

Hóa đơn Gemini \$500 là bài học đắt giá. Mức giảm 91% context theo sau đó tiết kiệm được nhiều hơn thế trong các tháng tiếp theo. Quản lý chi phí không chỉ là về việc không chi tiêu quá mức — mà là về việc xây dựng các hệ thống bền vững để vận hành ở quy mô.

Chương 9: Giám Sát AI Agent Trong Môi Trường Production

“Monitoring truyền thống hỏi: service có đang chạy không? Monitoring AI agent hỏi: service có đang chạy không, nó có đang làm đúng việc không, nó có đang làm hiệu quả không, và nó có sắp làm bạn phá sản không?”

Giám sát AI agent khó hơn giám sát các service truyền thống. Một web server thì hoặc là đang phản hồi hoặc là không. Một AI agent có thể đang chạy về mặt kỹ thuật — process khỏe, memory ổn, CPU bình thường — trong khi đồng thời đang làm gì đó hoàn toàn sai: bị kẹt trong vòng lặp, tiêu thụ token gấp 10 lần bình thường, thậm chí thất bại với mọi task, hoặc đang chạy hai instance trùng lặp gây ra các vòng lặp restart liên hoàn.

Chương này đề cập đến monitoring stack tôi đã xây dựng, các failure mode cụ thể bạn cần theo dõi, và những câu chuyện thực tế đã dạy tôi cần instrument những gì.

Những Gì Bạn Thực Sự Cần Giám Sát

Trước khi nói về công cụ, cần nói rõ “giám sát AI agent” có nghĩa là gì. Có năm hạng mục:

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

Hầu hết các team chỉ bắt đầu với health monitoring và thêm phần còn lại chỉ sau khi có sự cố. Cách thông minh hơn là instrument cả năm hạng mục từ ngày đầu tiên — overhead là tối thiểu, và thời gian tiết kiệm được khi debug sẽ hoàn vốn trong vài tuần.

Monitoring Stack

Stack production của tôi trên monitoring server:

```
Prometheus → scrapes metrics from all servers
Grafana    → dashboards and alerting
Loki       → log aggregation
Tempo      → distributed traces
Alertmanager → routes alerts to Telegram
```

Tất cả đều được bind vào VPN interface (10.10.0.0/24), không thể truy cập công khai.

Cấu Hình Prometheus

```
# /etc/prometheus/prometheus.yml

global:
  scrape_interval: 15s
  evaluation_interval: 15s
  external_labels:
    cluster: 'production'
    region: 'region-a-b'

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 (Region A - 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 (Region A - Cognee)
- job_name: 'server-2-node'
  static_configs:
    - targets: ['10.10.0.3:9100']
  relabel_configs:
    - target_label: server
      replacement: server-2

# Server-3 (Region B - 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

# Cloud ARM
- job_name: 'cloud-arm-node'
  static_configs:
    - targets: ['10.10.0.5:9100']
  relabel_configs:
    - target_label: server
      replacement: cloud-arm

```

Alert Rules Cho 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=~"agent-gateway.*|traefik|postgres|redis"})
      or
      time() - container_last_seen{name=~"agent-gateway.*|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 for more than 2 minutes."

  # 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 last
hour."

  # Memory usage too high
  - alert: AgentHighMemory
    expr: |
      container_memory_usage_bytes{name=~"agent-gateway.*|agent.*"}
      / container_spec_memory_limit_bytes{name=~"agent-gateway.*|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 its
memory limit."

  # CPU spike – possible runaway loop
  - alert: AgentCPUSpike
    expr: |
      rate(container_cpu_usage_seconds_total{name=~"agent-gateway.*|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 processes."

```

```

# 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."

```

Cấu Hình Alertmanager

```

# /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: Cảnh Báo Container Thời Gian Thực

Prometheus scrape theo chu kỳ 15 giây. Điều đó có nghĩa là một container có thể bị down đến 15 giây trước khi Prometheus biết được, và sau đó alert cần kích hoạt và được route qua Alertmanager trước khi bạn nhận được thông báo. Trong thực tế, độ trễ thông báo có thể là 2-5 phút.

Với các agent service quan trọng, đó là quá chậm. Tôi đã xây dựng một Docker events listener kích hoạt Telegram alert trong vòng dưới 5 giây.

```

#!/usr/bin/env python3
# /opt/agents/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 = {
    "agent-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 "i"
    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=agent-user
ExecStart=/home/agent-user/.venv/bin/python3 /opt/agents/scripts/docker-events-listener.py
Restart=always
RestartSec=10
EnvironmentFile=/home/agent-user/.env

[Install]
WantedBy=multi-user.target

```

Với cái này đang chạy, sự cố container sẽ tạo ra Telegram alert trong 3-5 giây, so với 2-5 phút thông qua Prometheus + Alertmanager.

Agent Mesh Monitor: Kiểm Tra Sức Khỏe Đa Server

Với các agent chạy trên nhiều server, bạn cần một hệ thống kiểm tra sức khỏe có thể nhìn xuyên qua ranh giới server. Tôi đã xây dựng cái mà tôi gọi là “agent mesh monitor” — một script nhẹ kiểm tra sức khỏe của mọi agent service từ một điểm quan sát trung tâm.

```
#!/bin/bash
# /opt/agents/scripts/agent-mesh-monitor.sh
# Runs on the 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"
  "cloud-arm:10.10.0.5"
)

# Services to check per server (name:port:path)
declare -A server_services
server_services["server-1"]="agent-gateway:3000:/health traefik:8080:/ping n8n:5678/ postgres:5432"
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["cloud-arm"]="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

# 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

```

Script Kiểm Tra Sức Khỏe Theo Từng Server

Mỗi server cũng chạy script kiểm tra sức khỏe cục bộ riêng. Đây là phiên bản đơn giản hóa cho thấy pattern:

Server-1: 22 services được kiểm tra

```
#!/bin/bash
# /opt/agents/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=(
    "agent-gateway"
    "traefik"
    "socket-proxy"
    "n8n"
    "n8n-postgres"
    "chatwoot-web"
    "chatwoot-sidekiq"
    "chatwoot-postgres"
    "chatwoot-redis"
    "api-proxy"
    "provider-manager"
    "postgres-main"
    "redis-main"
    "pgbouncer"
)

# Systemd services (8)
SYSTEMD_SERVICES=(
    "agent-gateway"
    "docker"
    "wg-quick@wg0"
    "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

```

Câu Chuyện Thực Tế #1: Kiểm Tra Sai Service

Ngày: Tháng 2 năm 2026 **Tác động:** Báo cáo sức khỏe sai trong nhiều tuần
Triệu chứng: Health check báo agent-gateway khỏe mạnh trong khi thực tế nó đã bị lỗi

Chuyện Gì Đã Xảy Ra

Tôi có một health check script trên Server-1 kiểm tra xem systemd service `agent-gateway` có đang chạy không. Nó trông ổn:

```

check_systemd_service "agent-gateway"
#  agent-gateway

```

Nhưng gateway agent tiếp tục hoạt động thất thường. Phản hồi Telegram không ổn định. Đôi khi các task hoàn thành bình thường, đôi khi hết thời gian. Health check liên tục báo xanh.

Sau khi đào sâu hơn, tôi tìm ra vấn đề: có hai service `agent-gateway` khác nhau.

```

# User-level systemd service (the one that actually mattered)
systemctl --user status agent-gateway
# ● agent-gateway.service - Orchestrating AI Agent
#   Loaded: loaded (/home/agent-user/.config/systemd/user/agent-gateway.service; enabled)
#   Active: failed (Result: exit-code) since ...

```

```
# System-level systemd service (what my check was hitting)
systemctl status agent-gateway
# ● agent-gateway.service - Agent Gateway (system stub)
#   Loaded: loaded (/etc/systemd/system/agent-gateway.service; enabled)
#   Active: active (running) since ... ← This was just a stub/wrapper
```

Health check script của tôi dùng `systemctl is-active agent-gateway` mà không có flag `--user`. Nó đang kiểm tra system-level stub (chỉ là một symlink hoặc wrapper), không phải user-level service nơi gateway agent thực sự chạy.

Cách Sửa

```
# WRONG: checks system-level service
systemctl is-active agent-gateway

# CORRECT: checks user-level service
systemctl --user is-active agent-gateway

# 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 "agent-gateway" "agent-user"
```

Bài Học Rút Ra

Khi bạn có cả system-level và user-level systemd service, monitoring script của bạn cần rõ ràng về scope nào đang kiểm tra. Mặc định (`systemctl status name`) kiểm tra system scope. Thêm `--user` và tùy chọn `-M user@` cho user-level service.

Ngoài ra: khi thứ gì đó “vượt qua health check” nhưng lại hoạt động thất thường, chính health check đó có thể sai. Hãy nghi ngờ monitoring của bạn trước khi cho rằng service ổn.

Câu Chuyện Thực Tế #2: Vòng Lặp Crash 102 Lần Restart

Ngày: Tháng 2 năm 2026 **Tác động:** Gateway không ổn định trong nhiều giờ

Triệu chứng: Gateway agent phản hồi không nhất quán, thỉnh thoảng có tin nhắn trùng lặp

Chuyện Gì Đã Xảy Ra

Tôi nhận thấy gateway agent đang gửi tin nhắn Telegram trùng lặp thỉnh thoảng. Cứ vài phút, phản hồi lại đến hai lần. Đôi khi các task dường như chạy hai lần. Đào vào log:

```
journalctl -u agent-gateway --no-pager -n 100 | grep "restart"
# Feb 07 21:30:01 systemd[1]: agent-gateway.service: Main process exited
# Feb 07 21:30:11 systemd[1]: agent-gateway.service: Start request repeated too quickly
# ... (repeated 102 times)
```

102 lần thử restart. Thứ gì đó đang khiến service crash liên tục, systemd tiếp tục restart nó, và mỗi lần restart sống sót đủ lâu sẽ nhận các tin nhắn đang chờ — nhưng instance trước đã bị crash đã xử lý một phần một số trong đó.

Nguyên Nhân Gốc Rễ: Hai Process Đang Chạy

```
pgrep -la "agent-gateway\|gateway"
# 768492 node /home/agent-user/.agent-gateway/gateway/index.js
# 771203 node /home/agent-user/.agent-gateway/gateway/index.js
```

Hai gateway process đang chạy đồng thời. Cả hai đều kết nối với cùng Telegram webhook, cả hai đều xử lý tin nhắn đến, và cả hai đều thực thi các task. Điều này gây ra: - Xử lý tin nhắn trùng lặp - Race condition trên shared state files - Port conflict gây crash (một process bind port mà cái kia đang dùng) - Vòng lặp crash → restart → conflict → crash

Điều Đó Xảy Ra Như Thế Nào

Một cron job được thiết kế để đảm bảo gateway đang chạy có race condition:

```
# Broken cron (runs every minute)
if ! systemctl is-active --quiet agent-gateway; then
    systemctl start agent-gateway
fi
```

Kiểm tra và start không phải là atomic. Trong tình huống service đang khởi động (chưa “active” nhưng không “failed”), cron sẽ thấy “not active” và phát lệnh `start` khác — dẫn đến hai lần start, hai process, hỗn loạn.

Cách Sửa

```
# Use systemctl's built-in restart logic instead of manual check-and-start
# This is atomic
systemctl restart agent-gateway

# 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 "agent-gateway/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 agent-gateway | cut -d= -f2)
        pgrep -f "agent-gateway/gateway" | grep -v "$systemd_pid" | xargs kill -TERM 2>/dev/null
        sleep 2
    fi

    systemctl is-active --quiet "$service" || systemctl start "$service"
}
```

Và thêm monitoring alert cho nhiều instance:

```
# prometheus rule
- alert: DuplicateAgentProcesses
  expr: |
    count by (instance) (
      process_cpu_seconds_total{job="agent-gateway"}
    ) > 1
  for: 1m
  labels:
    severity: critical
  annotations:
    summary: "Multiple gateway agent processes detected"
    description: "{{ $value }}" gateway agent processes running on {{ $labels.instance }}. Expected
1."
```

Bài Học Rút Ra

Phát hiện process trùng lặp cần phải rõ ràng trong monitoring của bạn. Health check tiêu chuẩn chỉ xác minh “có thứ gì đó đang chạy” — chúng không xác minh “đúng một thứ đang chạy.” Với agent service nơi thực thi trùng lặp gây hại thực sự (API call trùng lặp, race condition, data corruption), hãy thêm kiểm tra số lượng process chính xác.

```
# Quick check: how many instances of your agent are running?  
pgrep -c -f "your-agent-process-name"  
# Should be exactly 1
```

Neural Memory Audit: Khi Agent Có Sức Khỏe Riêng

AI agent sử dụng hệ thống bộ nhớ dài hạn (vector database, knowledge graph) cần hạng mục giám sát sức khỏe riêng. Hệ thống bộ nhớ có thể suy giảm theo thời gian theo những cách không hiện ra trong các infrastructure metric tiêu chuẩn.

Tôi đã thực hiện một cuộc kiểm tra chính thức về neural memory của gateway agent (hàng chục megabyte, hàng trăm memories vào thời điểm đó) và chấm điểm nó:

```
Neural Memory Audit Report  
Date: February 2026  
Agent: Orchestrating Gateway Agent
```

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.
```

Một memory điểm D có nghĩa là agent đang làm việc với thông tin lỗi thời, mâu thuẫn và không đầy đủ. Điều này không gây ra outage — agent tiếp tục chạy — nhưng nó gây ra suy giảm tinh tế trong chất lượng phản hồi và ra quyết định.

Các Memory Health Metric Cần Theo Dõi

```
# 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/agent-user/.agent-gateway/memory.db")
time.sleep(300) # Update every 5 minutes
```

Cấu Trúc Grafana Dashboard

Production Grafana của tôi có bốn dashboard:

Dashboard 1: Infrastructure Overview (Tổng Quan Hạ Tầng)

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 (Lịch Sử Alert)

- Recent alerts timeline
- Alert frequency by type
- MTTR (Mean Time to Recovery) by alert
- False positive rate

Tổng Hợp Log Với Loki

Logging tập trung là thiết yếu khi agent chạy trên nhiều server. Nếu không có nó, việc debug đòi hỏi SSH vào từng server riêng để đọc log.

```
# 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

# 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
```

Checklist Monitoring Hoàn Chỉnh

Dùng cái này khi onboarding một agent service mới:

Infrastructure Monitoring:

- [] node_exporter installed and scraping on VPN IP
- [] cadvisor installed for Docker metrics
- [] UFW rules allow Prometheus scrape from monitoring server (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
-

Tóm Tắt

Giám sát AI agent đòi hỏi vượt ra ngoài các infrastructure metric truyền thống. Bạn cần theo dõi:

1. **Health:** Agent có đang chạy không, và đúng một instance?
2. **Tokens:** Mức tiêu thụ có bình thường không, hay có thứ gì đó đang đốt quota?
3. **Errors:** API failures, rate limit, context limit — tất cả cần visibility
4. **Latency:** Suy giảm thời gian phản hồi là dấu hiệu cảnh báo sớm

5. **Cost:** Theo dõi chi tiêu thời gian thực ngăn ngừa hóa đơn bất ngờ

Docker events listener là bổ sung giá trị cao nhất vào stack này — độ trễ alert dưới 5 giây so với vài phút thông qua Prometheus. Nếu bạn không thực hiện gì khác từ chương này, hãy thực hiện cái đó.

Các câu chuyện thực tế ở đây có chung chủ đề: monitoring tiêu chuẩn báo xanh trong khi thực sự có gì đó sai. Giải pháp không phải là monitoring tốt hơn những thứ sai — mà là monitoring rõ ràng những thứ đúng. Kiểm tra process trùng lặp. Kiểm tra đúng systemd scope. Nghi ngờ health check báo khỏe mạnh trong khi hành vi thất thường.

Chương 10 đi sâu hơn vào debugging — phải làm gì khi alert kích hoạt và bạn cần tìm ra điều gì thực sự sai.

Chương 10: Debug AI Agent — Khi Mọi Thứ Đi Sai

“Debug một service truyền thống giống như tìm chìa khóa thất lạc trong nhà bạn. Debug một AI agent giống như tìm chìa khóa thất lạc trong một ngôi nhà mà các phòng tự sắp xếp lại trong khi bạn đang tìm kiếm.”

Debug AI agent là một lĩnh vực thực sự khác so với debug phần mềm truyền thống. Các bug truyền thống mang tính deterministic: với cùng một đầu vào, bạn nhận được cùng một đầu ra sai. Bạn có thể tái hiện chúng, bước qua chúng với debugger, và cô lập đúng dòng code sai.

Agent bug thì khác. Hành vi của agent phụ thuộc vào phản hồi của model, vốn mang tính xác suất. Cùng một prompt có thể tạo ra hành vi khác nhau trong các lần chạy khác nhau. Agent có thể đã sửa đổi môi trường của chính nó — thay đổi config file, di chuyển file, gửi tin nhắn — làm cho bối cảnh tái hiện khác với lúc bug xảy ra. Và khi bạn nhận ra có gì đó sai, agent có thể đã thực hiện hàng chục hành động mà bạn cần gỡ rối.

Chương này liệt kê các failure mode tôi đã gặp khi chạy AI agent trong production, với các phương pháp debug cụ thể cho từng loại.

Thay Đổi Tư Duy Khi Debug

Trước khi đi vào các failure cụ thể, cần thiết lập điều gì khác biệt trong agent debugging.

Các giả định debug truyền thống không áp dụng được cho agent:

“Tôi có thể tái hiện điều này một cách đáng tin cậy” — Không phải lúc nào cũng vậy. Hành vi model biến đổi. Nếu bug nằm ở cách agent diễn giải một tình huống mơ hồ, bạn có thể không thể kích hoạt lại nó một cách đáng tin cậy.

“Code là nguồn sự thật” — Agent có thể sửa đổi môi trường vận hành của chính nó. Config file, environment variable, state file — bất kỳ thứ nào trong số này có thể đã bị thay đổi bởi chính agent. Những gì bạn thấy trong codebase có thể không phản ánh những gì đang chạy khi bug xảy ra.

“Lỗi là tường minh” — Agent failure thường im lặng. Agent kết thúc, báo cáo thành công, và có gì đó tinh tế sai. Hoặc nó không bao giờ báo cáo gì cả.

“Hệ thống không có trạng thái giữa các lần chạy” — Agent thường duy trì trạng thái qua các session: hệ thống memory, context file, conversation history. Bug trong một session có thể đầu độc trạng thái cho các session tương lai.

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?)
```

Làm việc từ dưới lên. Xác minh infrastructure trước, rồi process health, rồi API behavior, rồi tool execution, rồi task outcomes. Hầu hết kỹ sư đi thẳng vào Level 5 và lãng phí thời gian debug hành vi agent trong khi vấn đề thực sự là Redis connection failure ở Level 1.

Failure Mode 1: Agent Sửa Đổi Chính Cơ Sở Hạ Tầng Của Nó

Tần suất: Không thường xuyên nhưng thảm khốc khi xảy ra **Phát hiện:** Gián đoạn service ngay sau agent task

Sự Cố

Agent orchestrating của chúng tôi cần cấu hình tích hợp Claude Code để route qua API proxy service. Task là: cập nhật cấu hình Claude Code để nó sử dụng proxy endpoint.

Agent nhìn vào các config file có sẵn, tìm thấy API proxy config tại `~/api-proxy/config.json`, và sửa đổi nó. Đây là file sai hoàn toàn — nó đã sửa đổi cấu hình nội bộ của proxy service thay vì client kết nối với proxy.

```
What the agent should have done:
  Edit ~/.claude.json → set ANTHROPIC_BASE_URL to the proxy endpoint

What the agent actually did:
  Edit ~/.api-proxy/config.json → changed the proxy's internal routing
  (Breaking the proxy that the agent itself was running through)
```

Kết quả: API proxy service bị cấu hình sai, ngừng route request đúng cách, và agent mất khả năng thực hiện API call — giữa task. Agent thậm chí không thể báo cáo điều đã xảy ra vì API nó dùng để báo cáo giờ đã bị hỏng.

Tại Sao Điều Này Xảy Ra

Agent đưa ra quyết định dựa trên tên file, đường dẫn và nội dung. Khi agent thấy `config.json` trong thư mục tên `api-proxy`, nó hợp lý suy ra đó là config file cần chỉnh sửa. Agent không vốn hiểu service nào là “chính nó” so với “service bên ngoài mà nó sử dụng.”

Phòng Ngừa

```
# 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:

- API Proxy Service (~/.api-proxy/) – The API proxy you run through.
  Modifying this breaks your own API access.
- Agent Gateway (~/.agent-gateway/gateway/) – Core agent runtime.
- systemd services: agent-gateway, api-proxy, 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 = [
    "~/.api-proxy/",
    "~/.agent-gateway/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}
```

Khôi Phục

```
# When an agent breaks its own infrastructure:

# 1. Find what changed recently
git -C ~/.api-proxy diff 2>/dev/null || \
  find ~/.api-proxy -newer /tmp/checkpoint -name "*.json" 2>/dev/null

# 2. Check service status
systemctl status api-proxy
journalctl -u api-proxy -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":
"ping"}]}'
```

Failure Mode 2: Lỗi Đánh Máy API Key và Lỗi Credential

Tần suất: Rất phổ biến **Phát hiện:** 401 Unauthorized errors, silent authentication failures

Sự Cố

Agent orchestrating của chúng tôi được cấu hình để sử dụng API key `agent-proxy-key-2026`. Sau một lần cập nhật cấu hình, nó ngừng hoạt động. Log hiển thị 401 error không liên tục, nhưng chỉ từ một số code path — những path khác có vẻ ổn.

Nguyên nhân gốc rễ là lỗi đánh máy tồn tại trong nhiều ngày: `gent-proxy-key-2026` thay vì `agent-proxy-key-2026`. Thiếu chữ `a` ở đầu.

```
# ~/.claude.json had:
{
  "apiKey": "gent-proxy-key-2026"
}

# Should have been:
{
```

```
    "apiKey": "agent-proxy-key-2026"
}
```

Lỗi đánh máy đặc biệt này khó phát hiện vì: 1. Key trông không sai rõ ràng — nó trông giống tên key 2. Một số code path dùng environment variable (có key đúng), trong khi các path khác dùng config file (có lỗi đánh máy) 3. 401 error xuất hiện không liên tục tùy thuộc vào code path nào thực thi

Debug Vấn Đề Xác Thực

```
#!/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 "APIProxy: %{http_code}\n" \
    http://localhost:8317/v1/models \
    -H "Authorization: Bearer $(grep -o 'agent[^\"]*' ~/.claude.json 2>/dev/null || echo
'KEY_NOT_FOUND')"
```

Pattern: Credential Không Nhất Quán Giữa Các File

Vấn đề liên quan: credential được cấu hình đúng ở một nơi nhưng lỗi thời/sai ở nơi khác.

```
# Find all places a specific key or token appears
grep -r "agent-proxy" ~ --include="*.json" --include="*.env" --include="*.yaml" 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: Lỗi Permission

Tần suất: Phổ biến **Phát hiện:** Thông báo lỗi chứa “Permission denied”, “EACCES”, “cannot write to”

Các Lỗi Permission Phổ Biến Trong Bối Cảnh Agent

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)

```

Xây Dựng Permission Diagnostic Tool

```

#!/bin/bash
# /opt/agents/scripts/check-agent-permissions.sh

AGENT_USER="${1:-agent-user}"
WORKSPACE="${2:-/home/agent-user/.agent-gateway/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 Limit

Tần suất: Rất phổ biến theo quy mô **Phát hiện:** HTTP 429 responses, “quota exceeded” errors

Vấn Đề Vertex AI 429

Trong quá trình import hàng loạt vào Cognition (186 trang tài liệu DevOps), quá trình import bắt đầu chạm vào Vertex AI rate limit. Embedding model (textembedding-gecko) có quota mặc định 5 RPM — 5 requests per minute. Script import đang nhấn vào nó ở 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"

Pattern Xử Lý Rate Limit

```

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))]

```

API Proxy Tự Động Chuyển Khi Hết Quota

Với tài khoản provider free-tier được quản lý qua API proxy service, việc hết quota là hành vi được mong đợi. Proxy xử lý nó tự động, nhưng bạn cần hiểu khi nào bạn đã hết tất cả tài khoản:

```

# Check current quota status across all accounts
journalctl -u api-proxy --no-pager --since "1 week ago" | \
    grep -E "quota|limit|switch|exhausted" | tail -20

# Count usage per account this week
journalctl -u api-proxy --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 Không Báo Cáo Tiến Trình

Tần suất: Rất phổ biến **Tác động:** Người dùng khó chịu, từ bỏ task, chẩn đoán sai failure

Vấn Đề

Một agent task chạy lâu mất 10-15 phút. Người dùng không thấy gì trong 10 phút, cho rằng agent bị kẹt hoặc crash, và hoặc là kill nó hoặc bắt đầu một task trùng lặp. Khi task gốc hoàn thành 5 phút sau, bạn có kết quả trùng lặp hoặc workflow bị hỏng.

Đây là UX bug, không phải technical bug — nhưng nó gây ra vấn đề kỹ thuật thực sự ở downstream.

Pattern Ngăn Ngừa Điều Này

```

# 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

```

Với hệ thống agent cụ thể, hãy triển khai “heartbeat” output xác nhận agent vẫn còn sống ngay cả khi không có gì mới để báo cáo:

```

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: Vượt Quá Context Limit Giữa Task

Tần suất: Phổ biến với sub-agent chạy lâu **Phát hiện:** Kết thúc task đột ngột, lỗi chứa “context_length_exceeded” hoặc “200000”

Vấn Đề

Context window của Claude là 200K token. Một sub-agent làm việc trên codebase lớn, hoặc task liên quan đến đọc nhiều file, có thể chạm giới hạn này giữa task. Khi đó, task kết thúc đột ngột — thường không có thông báo lỗi hữu ích được truyền lại cho orchestrator.

Typical scenario:

1. Orchestrator spawns sub-agent: "Analyze all 89 TypeScript files in src/ and generate documentation"
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

Phòng Ngừa: Quản Lý Context Budget

```

# 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}

```

Pattern Chunking

Với các task về bản chất đòi hỏi nhiều context hơn khả năng chứa trong một window, hãy dùng chunking tương minh:

```
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

Tần suất: Không thường xuyên nhưng ngấm ngấm **Phát hiện:** Suy giảm hiệu suất, hành vi không mong đợi, tên model trong API response

Sự Cố

API proxy service được cấu hình để phục vụ Claude Opus request, nhưng do vấn đề quota tài khoản, nó đang âm thầm downgrade request xuống Claude Sonnet. Config file ghi `claude-opus-4-5-thinking`, API response báo `claude-sonnet-4-5-thinking`, và agent đang hành động theo đó — kém capable hơn mong đợi.

```
# Check what model the proxy is actually serving
journalctl -u api-proxy --no-pager -n 20 | grep "model"
# Using provider account ... model=claude-sonnet-4-5-thinking
# Expected: model=claude-opus-4-5-thinking
```

Mismatch xảy ra vì: 1. Opus có tốc độ tiêu thụ quota cao hơn 2. Tài khoản chính đã chạm quota Opus 3. Proxy fallback xuống Sonnet mà không thông báo rõ ràng cho caller 4. Agent tiếp tục chạy, có vẻ hoạt động nhưng với chất lượng reasoning suy giảm

Phát Hiện Model Mismatch

```
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")
```

Câu Chuyện Thực Tế: SSR CSS Hash Mismatch

Ngày: Tháng 2 năm 2026 **Triệu chứng:** Website hiển thị style bị hỏng sau deployment **Thời gian chẩn đoán:** 45 phút (đáng ra chỉ cần 5 phút)

Chuyện Gì Đã Xảy Ra

Một ứng dụng Next.js được deploy. Build chạy thành công trên Server-2. App có thể truy cập. Nhưng styling trông hoàn toàn bị hỏng — không có CSS, chỉ là HTML thô.

Lỗi trong browser console:

```
Prop className did not match.
Server: "styles__Container-abc123"
Client: "styles__Container-xyz789"
```

CSS class hash mismatch là vấn đề hydration Next.js kinh điển. Nhưng nguyên nhân không nằm trong code — mà nằm trong quá trình deployment.

Chuỗi sự kiện: 1. Build mới được tạo trên Server-2 (build ID: abc123) 2. Nginx đang phục vụ build cũ (build ID: xyz789) từ deployment trước 3. Server-side rendering dùng build cũ của server → tạo class xyz789 4. Client tải JS bundle mới → mong đợi class abc123 5. Mismatch ở mỗi lần tải trang

Cách Sửa

```
# 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
```

Bài Học Rút Ra

Ứng dụng server-side rendering giữ compiled asset trong bộ nhớ. Deployment mới không tự động reload chúng. Điều này rõ ràng khi nhìn lại nhưng dễ bỏ qua khi bạn đang tập trung vào quá trình build hơn là server đang chạy.

Thêm điều này vào 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
```

Câu Chuyện Thực Tế: Config Sync Trên Ba Server

Ngày: Tháng 2 năm 2026 **Triệu chứng:** Hành vi agent không nhất quán tùy thuộc vào server nào xử lý request **Thời gian chẩn đoán:** Nhiều ngày với những phân nản không liên tục

Chuyện Gì Đã Xảy Ra

Gateway agent được cập nhật để dùng cấu hình model mới. Cập nhật được áp dụng cho Server-1. Server-2 và Server-3 vẫn có cấu hình cũ. Vì request có thể được xử lý bởi bất kỳ server nào thông qua load balancer, hành vi không nhất quán — đôi khi model mới, đôi khi model cũ.

```
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
...
```

Thách thức debug: hành vi không nhất quán trông giống model không đáng tin hoặc bug trong logic agent. Mất vài ngày để nhận ra vấn đề là config drift giữa các server.

Chẩn Đoán

```
# Compare config files across all servers
for server in server1 server2 server3; do
    echo "=== $server ==="
    ssh $server "cat ~/.agent-gateway/agent-gateway.json | python3 -m json.tool | head -20"
done

# Find differences
diff \
    <(ssh server1 "cat ~/.agent-gateway/agent-gateway.json | python3 -m json.tool") \
    <(ssh server2 "cat ~/.agent-gateway/agent-gateway.json | python3 -m json.tool")
```

Phòng Ngừa: Config Sync Script

```
#!/bin/bash
# /opt/agents/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=(
    "~/.agent-gateway/agent-gateway.json"
    "~/.claude/settings.json"
)
```

```

    "~/api-proxy/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" "agent-user@${target_ip}:${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 ${target_name}..."
    ssh "agent-user@${target_ip}" "systemctl --user restart agent-gateway 2>/dev/null; \
        docker restart api-proxy 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 "agent-user@${server_ip}" "md5sum ~/.agent-gateway/agent-gateway.json 2>/dev/
null" | cut -d' ' -f1)
    echo "${server_name}: ${checksum}"
done

```

Giải Pháp Dài Hạn Tốt Hơn: Config Tập Trung

Với agent deployment đa server, hãy lưu trữ cấu hình trong store tập trung thay vì sync file thủ công:

```

# 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")

```

Câu Chuyện Thực Tế: Etcd Split-Brain

Ngày: Tháng 2 năm 2026 **Triệu chứng:** 22 sự kiện failover trong một ngày, hành vi service không nhất quán **Thời gian giải quyết:** 4 giờ

Chuyện Gì Đã Xảy Ra

Một etcd cluster (3 node) chạy trên các server phát triển config mismatch. Server-1 có cluster ID khác với Server 2 và 3. Điều này gây ra tình trạng split-brain: cluster không thể thống nhất về leader, dẫn đến bầu cử leader và failover liên tục.

```

# 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

```

Cách Sửa

```

# 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=http://
10.10.0.1:2380"
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
```

Câu Chuyện Thực Tế: Lỗi Reasoning Parameter

Ngày: Tháng 2 năm 2026 **Triệu chứng:** Google API call trả về HTTP 400 errors **Thời gian chẩn đoán:** ~2 giờ

Chuyện Gì Đã Xảy Ra

Sau khi cập nhật cấu hình agent để dùng extended thinking (sử dụng parameter `reasoning_effort` cho Claude), agent bắt đầu fail trên các request được route đến Google Gemini model. Lỗi rất khó hiểu:

```
HTTP 400 Bad Request
{
  "error": {
    "code": 400,
    "message": "Invalid JSON payload received. Unknown name 'reasoning_effort'",
    "status": "INVALID_ARGUMENT"
  }
}
```

Agent framework đang truyền `reasoning_effort` cho mọi model request, vì nó được đặt global trong cấu hình. Claude hiểu `reasoning_effort`. Google Gemini thì không — nó trả về 400 error khi nhận parameter không xác định.

```
# 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
    )
```

Cách Sửa

```
# 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
```

```
)
```

Câu Chuyện Thực Tế: Claude Code Kẹt Ở Trust Dialog

Ngày: Tháng 2 năm 2026 **Triệu chứng:** Claude Code session không phản hồi, task không thực thi **Thời gian chẩn đoán:** 30 phút

Chuyện Gì Đã Xảy Ra

Claude Code đang chạy trong một tmux session (`claude-code`), được cấu hình để tự động xử lý task từ agent orchestrating. Task đang được gửi đến session nhưng không có phản hồi. Session có vẻ active nhưng idle.

```
# Capture what's actually showing in the tmux session
tmux capture-pane -p -t claude-code
# OUTPUT:
# |
# |         Trust this folder?
# |
# | /home/agent-user
# |
# | Do you trust the files in this folder?
# |
# | 1. Yes, trust /home/agent-user and all subfolders
# | 2. Yes, trust /home/agent-user (this session only)
# | 3. No (recommended)
# |
# |
```

Claude Code đã mở tới thư mục `/home/agent-user` (chưa được trust trước đó) và đang chờ input của người dùng trên trust prompt. Vì nó đang chạy trong automated tmux session, không có người để click “Yes.” Tất cả task đến đang xếp hàng sau dialog chưa được trả lời này.

Cách Sửa (Ngắn Hạn)

```
# 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
```

Cách Sửa (Dài Hạn)

```
// ~/.claude.json – Pre-trust directories that Claude Code will work in
{
  "projects": {
    "/home/agent-user": {
      "hasTrustDialogAccepted": true
    },
    "/home/agent-user/.agent-gateway/workspace": {
      "hasTrustDialogAccepted": true
    },
    "/home/agent-user/projects": {
      "hasTrustDialogAccepted": true
    }
  }
}
```

Thêm trust dialog check vào 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
}
```

Câu Chuyện Thực Tế: Sub-Agent Silent Failure

Ngày: Tháng 2 năm 2026 **Triệu chứng:** Task báo hoàn thành, thư mục không tồn tại **Thời gian chẩn đoán:** 45 phút

Chuyện Gì Đã Xảy Ra

Agent orchestrating của chúng tôi spawn một sub-agent (Deploy_Expert) để deploy Firecrawl trên Server-3. Sub-agent chạy, có vẻ hoàn thành, và báo cáo thành công. Khi tôi sau đó cố truy cập Firecrawl tại vị trí mong đợi:

```
ls /root/firecrawl
# ls: cannot access '/root/firecrawl': No such file or directory
```

Thư mục không tồn tại. Việc deployment chưa bao giờ xảy ra. Nhưng sub-agent đã báo cáo thành công.

Điều Thực Sự Đã Xảy Ra

Sub-agent đã SSH vào Server-3, chạy một vài lệnh, gặp lỗi (có thể là vấn đề permission hoặc network), và sau đó — quan trọng — không propagate lỗi đó lên agent orchestrating. Nó trả về thông báo thành công dựa trên các lệnh nó đã chạy thành công, không phải dựa trên việc deployment có thực sự hoạt động không.

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

Phòng Ngừa: Bước Xác Minh Trong Sub-Agent Task

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"agent-user@{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

```

Debugging Checklist

Khi có gì đó sai với AI agent, hãy làm việc qua checklist này theo thứ tự:

```

## 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 API proxy service 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?
```

Tóm Tắt

Agent debugging đòi hỏi cách tiếp cận hoàn toàn khác so với debug phần mềm truyền thống. Các nguyên tắc chính:

1. **Làm việc từ dưới lên** — xác minh infrastructure trước khi nghi ngờ hành vi agent
2. **Không tin vào agent self-report** — xác minh kết quả độc lập, đặc biệt là với deployment
3. **Báo cáo tiến trình tường minh là bắt buộc** — im lặng trông giống thất bại với người dùng
4. **Kiểm tra process trùng lặp tường minh** — health check tiêu chuẩn sẽ không phát hiện chúng
5. **Xác định rõ systemd check của bạn** — user-level vs system-level service là khác nhau
6. **Lọc parameter theo provider** — những gì Claude chấp nhận, Gemini có thể không
7. **Config drift giữa các server gây ra bug không liên tục** — kiểm tra tất cả server, không chỉ một
8. **Giám sát tmux session** — orphan session âm thầm đốt quota và có thể chạy công việc trùng lặp
9. **Context limit gây ra truncation im lặng** — triển khai budget management trước khi chạm tường
10. **Bảo vệ cơ sở hạ tầng quan trọng khỏi agent** — agent cần ràng buộc tường minh về những gì chúng có thể sửa đổi

Chủ đề chung trong tất cả các câu chuyện thực tế: agent đang chạy, về mặt kỹ thuật. Nó đang làm gì đó. Những gì nó đang làm chỉ là sai, theo những cách mà monitoring “có nó chạy không?” tiêu chuẩn không thể phát hiện.

Đó là lý do chương monitoring đến trước chương này. Bạn cần observability sẵn sàng trước khi cần debug. Khi thứ gì đó bị hỏng, bạn muốn log, metric, và alert history sẵn sàng để nói cho bạn biết điều gì đã xảy ra — không chỉ điều gì đang xảy ra bây giờ.

Các agent sẽ làm những điều bạn không mong đợi. Hãy xây dựng các hệ thống để bắt nó.

PHẦN 4: CHỦ ĐỀ NÂNG CAO

Chương 11: Tăng Cường Bảo Mật Cho Cơ Sở Hạ Tầng AI Agent

“Agent đã in token root Vault của chúng tôi ngay trong cuộc trò chuyện. Ba mươi giây sau tôi đang khẩn cấp thu hồi nó trong khi soạn báo cáo sự cố.”

Ngày AI Trở Thành Lỗ Hổng Bảo Mật Lớn Nhất Của Chúng Tôi

Nó bắt đầu như một phiên debug.

Chúng tôi có một kỹ sư mới onboarding, và tôi đang hướng dẫn anh ấy cách AI agent của chúng tôi có thể hỗ trợ các tác vụ hạ tầng. Chúng tôi yêu cầu agent giúp chẩn đoán vấn đề kết nối Vault. Agent — cố gắng hữu ích — yêu cầu chúng tôi dán các environment variable hiện tại để nó có thể phân tích cấu hình.

Chúng tôi đã làm vậy.

Root token xuất hiện trong chat history. Không bị che. Trong một cuộc trò chuyện được sync lên cloud. Trong một log file được rotate nhưng không được mã hóa. Trong một session mà ba người khác có quyền truy cập.

Đó là khoảnh khắc chúng tôi nhận ra rằng AI agent trong cơ sở hạ tầng production đòi hỏi mô hình bảo mật hoàn toàn khác so với các công cụ truyền thống. Các kỹ sư của bạn biết không nên paste credential vào Slack. Nhưng họ có biết không nên chia sẻ environment context với AI agent ghi log mọi thứ không?

Chương này đề cập đến những gì chúng tôi học được theo cách khó: cách tăng cường cơ sở hạ tầng AI agent, khóa chặt các service xung quanh, và xây dựng tư thế bảo mật coi AI là vector tấn công tiềm năng, không chỉ là công cụ năng suất.

Threat Model Đã Thay Đổi

Bảo mật hạ tầng truyền thống giả định rằng con người mắc lỗi. Bạn xây dựng guardrail — MFA bắt buộc, IAM role least-privilege, secret scanning trong CI/CD — và đào tạo nhóm của bạn.

AI agent phá vỡ mô hình này theo nhiều cách:

Agent tổng hợp context. Một kỹ sư có thể không nhận ra họ có quyền truy cập cả mật khẩu database và danh sách production server. Agent làm việc một task sẽ tự nhiên kéo cả hai mảnh thông tin lại với nhau, tăng blast radius của bất kỳ sự xâm phạm nào.

Agent làm lộ thông tin ẩn. Agent hỏi những environment variable nào đã được set. Không có kỹ sư nào nghĩ đến việc hỏi điều đó trong phiên debug. Agent làm vậy vì đó là bước logic tiếp theo. Câu trả lời bao gồm

```
VAULT_TOKEN=s.xxxxxxxx.
```

Agent để lại dấu vết. Mọi cuộc trò chuyện, mọi tool call, mọi kết quả trung gian đều được ghi log ở đâu đó. Trong thiết lập của chúng tôi, điều đó có nghĩa là chat history trong application database, tool call log trong filesystem, và API request log tại provider. Ba nơi một credential bị rò rỉ có thể tồn tại.

Agent có thể bị prompt. Kẻ tấn công có thể ảnh hưởng đến những gì agent làm — thông qua một issue độc hại, một config file được tạo thủ công, một input source bị xâm phạm — có thể hướng agent để exfiltrate dữ liệu hoặc thực hiện hành động phá hoại.

Hãy xem qua các bước tăng cường bảo mật chúng tôi đã triển khai sau sự cố.

Phần 1: Quản Lý Vault Token

Vấn Đề Root Token

Vault root token không bao giờ nên được dùng trong production. Dứt khoát. Root token bỏ qua tất cả policy — nó có thể làm bất cứ điều gì. Chúng tôi đã dùng nó trong quá trình setup ban đầu và, qua một loạt quyết định “tạm thời”, nó đã ở lại trong environment variable của chúng tôi sáu tháng.

Khi agent làm lộ nó, chúng tôi phải coi nó như đã bị xâm phạm.

Xử lý khẩn cấp:

```
# 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
```

Cách đúng để dùng Vault trong production:

Không bao giờ dùng root token cho công việc vận hành. Tạo AppRole với policy tối thiểu.

```
# 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
```

Agent nhận một token ngắn hạn. Nếu nó xuất hiện trong chat log, nó đã hết hạn trước khi ai đó có thể sử dụng nó.

Tự Động Hóa Xoay Token

```
#!/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)"
```

Quy Trình Vault Seal/Unseal

Vault trong production nên được auto-unseal bằng cloud KMS (AWS KMS, GCP Cloud KMS, Azure Key Vault) hoặc được khởi tạo với Shamir secret sharing trên nhiều key holder.

Chúng tôi dùng Shamir với năm key holder, yêu cầu ba để 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
```

Để auto-unseal với AWS KMS:

```
# vault.hcl
seal "awskms" {
  region      = "ap-southeast-2"
  kms_key_id = "arn:aws:kms:ap-southeast-2:123456789:key/your-key-id"
}
```

Phần 2: Tăng Cường Network — Các Port Không Bao Giờ Nên Công Khai

Phát Hiện

Trong quá trình audit bảo mật, chúng tôi chạy port scan từ IP bên ngoài đối với server của chúng tôi.

```
nmap -sV -p 1-65535 --open your-server-ip
```

Kết quả thật kinh hoàng. Port 9100 (Prometheus Node Exporter), 9187 (PostgreSQL Exporter), 9081 (một số service nội bộ), 5432 (chính PostgreSQL trên server-3), và 6379 (Redis) đều có thể tiếp cận từ internet công cộng.

Ai đó có thể scrape metric của chúng tôi — CPU, memory, disk, query statistics, slow query log. Ai đó có thể cố kết nối trực tiếp vào PostgreSQL. Ai đó có thể kết nối vào Redis không cần xác thực (chúng tôi đã set `requirepass`, nhưng vẫn vậy).

Server-3 đã cài UFW nhưng **chưa bao giờ bật**. Nó đã được provisioned sáu tháng trước và không ai kiểm tra.

```
# The horrifying discovery
ssh server-3
sudo ufw status
# Status: inactive
```

Bật UFW Từ Đầu (Không Khóa Bản Thân Ra Ngoài)

Đây là chuỗi lệnh sẽ giúp bạn tránh thảm họa “Tôi bật UFW và giờ không SSH vào được” kinh điển.

```
#!/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
```

Bind Service Nhảy Cắm Vào VPN Interface

UFW rule là một lớp. Bind service vào VPN interface là lớp khác, tốt hơn.

Với PostgreSQL trong `postgresql.conf` :

```
# Only listen on VPN interface and localhost – never on public interface
listen_addresses = '127.0.0.1,10.8.0.10'
```

Với Redis trong `redis.conf` :

```
bind 127.0.0.1 10.8.0.10
```

Với Prometheus Node Exporter trong `systemd`:

```
[Service]
ExecStart=/usr/local/bin/node_exporter \
  --web.listen-address="10.8.0.10:9100"
```

Nếu service chỉ lắng nghe trên VPN interface, một cấu hình sai firewall không thể expose nó. Defense in depth.

Cloud ARM Server: Dọn Dẹp NFS/rpcbind

Cloud ARM database server của chúng tôi có `rpcbind` và NFS service đang chạy — tàn dư của cấu hình được copy-paste từ server khác cần NFS mount. Chúng tôi đã không cần NFS ở đây hơn một năm.

```
# 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"
```

Quy tắc: audit mọi port đang lắng nghe trên mọi server. Nếu bạn không biết tại sao nó ở đó, hãy tìm hiểu. Nếu bạn không cần nó, hãy kill nó.

Phần 3: Tăng Cường SSH

PermitRootLogin và PasswordAuthentication

Load balancer của chúng tôi là server cũ nhất trong fleet. Khi chúng tôi kiểm tra SSH config của nó:

```
grep -E '(PermitRootLogin|PasswordAuthentication|PubkeyAuthentication)' /etc/ssh/sshd_config
```

Output hiển thị các giá trị mặc định — có nghĩa là root login được phép và password authentication được bật. Cấu hình mặc định là kẻ thù của bảo mật.

```
# /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
```

Xoay SSH Key Khi Onboarding Máy Mới

Khi chúng tôi provisioned server mới, chúng tôi có thói quen xấu: copy `authorized_keys` từ server hiện có. Nhanh, dễ, và là ác mộng bảo mật. Key cũ từ các kỹ sư đã nghỉ việc đang tích lũy.

```
#!/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"
```

Flag `exclusive=yes` trong Ansible rất quan trọng — nó xóa tất cả key không có trong file của bạn. Không có nó, bạn đang thêm key, không thay thế chúng.

Restart Service Sau Khi Xoay Password

Chúng tôi học bài học này khi xoay PostgreSQL password qua Vault và application tiếp tục hoạt động — trong bốn mươi phút, cho đến khi các kết nối connection pool cũ đóng lại. Sau đó mọi thứ bị crash.

Service giữ kết nối mở. Xoay password không chấm dứt kết nối hiện có. Bạn cần hoặc:

1. Gracefully drain và restart service, hoặc
2. Dùng `pg_terminate_backend` để kill kết nối phía database

```
#!/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
connected"
    ;;
  "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"
```

Thêm xoay password vào runbook và bao gồm bước restart. Làm nó bắt buộc, không tùy chọn.

Phần 4: Credential Plaintext Trong Config File

Patroni.yml — Mật Khẩu Phơi Bày

Patroni quản lý PostgreSQL high availability. Config file của nó bao gồm database credential. Chúng tôi tìm thấy `patroni.yml` với:

- Quyền world-readable (`-rw-r--r-- root root`)
- Mật khẩu plaintext cho replication user, superuser, và monitoring user
- Được lưu trữ trong git repository “để backup”

```
# 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: MyR3plicat10nP@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

```

Cách sửa có hai phần: sửa quyền file, và dùng 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

```

Về lâu dài, hãy thay thế file `.env` bằng Vault Agent hoặc envconsul, vốn kéo secret tại runtime và inject chúng như environment variable mà không chạm đến disk.

Credential Hardcode Trong Application Code

Chúng tôi tìm thấy credential hardcode trong một background job runner. Service kết nối với external API provider bằng credential đã được “tạm thời” hardcode tám tháng trước và không bao giờ chuyển sang environment variable.

```
# BEFORE – credentials hardcoded in source
class ApiProviderClient:
    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.provider.io/v2"

# AFTER – credentials from environment
import os

class ApiProviderClient:
    def __init__(self):
        self.api_key = os.environ["API_PROVIDER_KEY"]
        self.api_secret = os.environ["API_PROVIDER_SECRET"]
        self.base_url = os.environ.get(
            "API_PROVIDER_BASE_URL",
            "https://api.provider.io/v2"
        )

    if not self.api_key or not self.api_secret:
        raise ValueError(
            "API_PROVIDER_KEY and API_PROVIDER_SECRET must be set"
        )
```

Sau khi sửa code, hãy xoay credential ngay lập tức. Coi bất kỳ credential nào đã từng ở trong source code là đã bị xâm phạm.

```
# 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
```

Phần 5: Docker Security — Vấn Đề docker.sock

Traefik và docker.sock

Traefik là reverse proxy tự động khám phá Docker container thông qua Docker socket. Thiết lập kinh điển mount `/var/run/docker.sock` trực tiếp vào container Traefik.

```
# INSECURE – Traefik with direct socket access
services:
  traefik:
    image: traefik:v3.0
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro # DANGEROUS
```

Bất kỳ ai có thể thực thi code bên trong container Traefik giờ có quyền truy cập Docker daemon đầy đủ. Họ có thể liệt kê tất cả container, đọc environment variable (chứa secret), khởi động container mới với `--privileged`, mount đường dẫn filesystem host — container escape đầy đủ.

Cách sửa là `docker-socket-proxy`, đứng giữa Traefik và Docker socket và thực thi bề mặt API tối thiểu.

```
# 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

```

Giờ Traefik không thể thực thi lệnh bên trong container, không thể đọc environment variable từ container khác, và không thể tạo container mới. Nó chỉ có thể đọc routing label mà nó cần.

Phần 6: HAProxy Stats – Mật Khẩu Mặc Định Không Ai Thay Đổi

HAProxy có giao diện statistics tích hợp. Cấu hình của chúng tôi có:

```
stats auth admin:admin
```

Trang stats này phơi bày: số kết nối active mỗi backend, session rate, error rate, trạng thái sức khỏe server, và địa chỉ IP backend server. Không phải credential trực tiếp, nhưng đủ intelligence vận hành để giúp kẻ tấn công hiểu topology của bạn.

```

# 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
```

Mật khẩu vào Vault. Bạn lấy nó tại deploy time và substitute vào 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
```

Phần 7: Vault Policy Cho Agent Access

AI agent cần quyền truy cập Vault cho các tác vụ vận hành hợp lệ. Nhưng nó không nên có quyền truy cập mọi thứ. Đây là cấu trúc policy chúng tôi dùng:

```
# 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

```

Phần 8: AI Agent Security Checklist

Sau tất cả những gì chúng tôi đã trải qua, đây là checklist chúng tôi chạy trước khi deploy bất kỳ AI agent nào chạm vào hạ tầng.

Trước Khi Deploy

- [] Agent chạy với dedicated service account (không phải root, không phải shared account)
- [] Service account có minimal filesystem permission
- [] Vault AppRole được tạo với least-privilege policy
- [] Agent token TTL <= 2 giờ
- [] Tất cả secret trong Vault hoặc environment variable – không có trong code hoặc config
- [] Docker socket access qua docker-socket-proxy nếu cần
- [] Truy cập mạng bị hạn chế: agent chỉ có thể tiếp cận service nó cần
- [] Lưu trữ conversation/log được mã hóa at rest
- [] Chính sách lưu giữ log được xác định (và ngắn – log chứa context là nhạy cảm)

Baseline Hạ Tầng

- [] UFW bật trên TẤT CẢ server (xác minh: sudo ufw status)
- [] PermitRootLogin no trên TẤT CẢ SSH config
- [] PasswordAuthentication no trên TẤT CẢ SSH config
- [] Tất cả database port được bind vào VPN interface
- [] Tất cả metric exporter port được bind vào VPN interface
- [] HAProxy stats dùng credential không mặc định từ Vault
- [] Patroni.yml dùng environment variable substitution
- [] Quyền patroni.yml: 600, owned by postgres:postgres
- [] Không có service không dùng đang chạy (NFS, rpcbind, v.v.)
- [] Traefik dùng docker-socket-proxy

Vận Hành

- [] Xoay Vault token được tự động hóa
- [] Quy trình xoay SSH key được document và tested
- [] Runbook bao gồm bước restart service sau khi xoay credential
- [] Secret scanning chạy trong CI/CD
- [] Port scan định kỳ được lên lịch (hàng tuần từ IP bên ngoài)
- [] Kế hoạch ứng phó sự cố đề cập đến tình huống "AI agent lộ credential"

Bài Học Rút Ra

Agent không có nhận thức về bảo mật. Nó sẽ trình bày bất kỳ thông tin nào giúp nó hoàn thành task. Nếu debug vấn đề kết nối Vault đòi hỏi hiển thị token, nó sẽ hiển thị token. Bạn không thể đào tạo AI để cẩn thận hơn — bạn phải làm cho môi trường an toàn.

Cấu hình mặc định là kẻ thù của bạn. Mỗi service chúng tôi tìm thấy với mật khẩu mặc định, cấu hình mặc định, hoặc không có firewall đều được thiết lập nhanh dưới áp lực thời gian. “Tôi sẽ bảo mật đúng cách sau” là câu nói tốn kém nhất trong hạ tầng.

Defense in depth không phải tùy chọn. UFW rule VÀ network binding VÀ VPN-only routing. AppRole policy VÀ short TTL VÀ token rotation. Nhiều lớp, vì bất kỳ lớp đơn nào cuối cùng cũng sẽ thất bại.

Audit mọi thứ, thường xuyên. Server với UFW không hoạt động đã như vậy sáu tháng. Không ai kiểm tra vì không ai có quy trình định kỳ để kiểm tra. Thêm `ufw status`, `ss -tlnp`, và `sshd -T | grep -E '(PermitRoot|PasswordAuth)'` vào audit script hạ tầng thường xuyên của bạn.

Xoay sớm, xoay thường xuyên. Khi nghi ngờ credential có thể bị lộ, hãy xoay nó. Chi phí xoay credential không thực sự bị lộ là một giờ làm việc. Chi phí không xoay một cái thực sự đã bị lộ là một sự cố, một cuộc kiểm tra, và có thể là một thông báo vi phạm.

Sự cố root token thật đáng xấu hổ. Nhưng nó buộc chúng tôi phải sửa sáu tháng nợ bảo mật tích lũy trong một tuần. AI agent, với sự hữu ích ngẫu nhiên của nó, đã giúp chúng tôi một việc.

Tiếp theo: Chương 12 — AI Image Generation ở Quy Mô Lớn: điều gì xảy ra khi bạn yêu cầu AI tạo ảnh cho gần 300 bài blog và chạm hạn mức quota giữa chừng.

Chương 12: AI Image Generation ở Quy Mô Lớn

“Chúng tôi chạm hạn mức quota ở bài số 147 trong gần 300 bài. Batch đã chạy được bốn tiếng. Tôi nhìn chăm chăm vào thông báo lỗi và nghĩ: lẽ ra mình phải test trên mười bài trước.”

Vấn Đề Xử Lý Hàng Loạt Ở Quy Mô Lớn

Mỗi bài blog cần một featured image. Chúng tôi có gần 300 bài blog sắp được đăng như một phần của content migration, và không có ảnh nào. Designer báo giá ba tuần và \$4,000. Chúng tôi có năm ngày và một Vertex AI quota.

Đề xuất nghe có vẻ đơn giản: dùng Gemini để tạo ảnh hàng loạt, review chúng, đăng. Bốn tiếng làm việc, không phải ba tuần.

Thực tế xảy ra là hai tuần chiến đấu với quota, viết lại script, thử nghiệm delimiter, và bài học khó về khoảng cách giữa “AI có thể tạo ảnh” và “AI có thể tạo ảnh chuyên nghiệp ở quy mô lớn.”

Chương này đề cập đến trải nghiệm production thực tế: các lần chạm quota, hot-swapping, checkpointing, và khoảnh khắc chúng tôi nhận ra AI-generated logo là ngổ cụt.

Kiến Trúc: Xử Lý Hàng Loạt Ở Quy Mô Lớn

Điểm Xuất Phát: Những Gì Chúng Tôi Có

Chúng tôi có một CSV các bài blog với title, slug, và meta description từ client project. Chúng tôi cần:

1. Tạo image prompt phù hợp từ post metadata
2. Gọi Vertex AI Imagen để tạo ảnh
3. Lưu với filename đúng
4. Theo dõi tiến độ để có thể resume nếu có gì đó thất bại

Script ban đầu có 80 dòng Python và hoạt động hoàn hảo với mười bài đầu tiên.

```
#!/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",
        )
    except:
        return False
    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()

```

Cái này hoạt động cho đến bài 147.

Chạm Quota Giữa Batch

Lỗi

```
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'.
```

Bốn tiếng làm việc. 147 ảnh đã tạo. 142 ảnh còn lại.

Quota mặc định Vertex AI cho image generation khá thận trọng — đủ cho development và testing, không đủ cho một batch job gần 300 bài.

Lựa Chọn 1: Yêu Cầu Tăng Quota (Mất Vài Ngày)

```
# 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
```

Chúng tôi cần ảnh trong năm ngày. Chờ phê duyệt quota không phải lựa chọn.

Lựa Chọn 2: Hot-Swap API Key Trên Nhiều Project

Con đường nhanh: tạo Google Cloud project thứ hai, bật Vertex AI API, và chuyển sang nó khi project đầu tiên chạm quota.

```
# 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)

```

Với ba project, mỗi project có quota 10 requests per minute, chúng tôi có hiệu quả 30 requests per minute. Batch đã bị dừng ở bài 147 chạy lại trong vòng mười phút.

Thảm Họa Delimiter

Lỗi Xuất Hiện Như Thế Nào

Title bài blog chứa đủ loại ký tự. Một trong số bài có title:

```
“PostgreSQL vs MySQL vs SQLite | Which Database Should You Choose?”
```

CSV của chúng tôi dùng `|` làm column delimiter.

```
# The CSV that caused problems
title|slug|meta_description
PostgreSQL vs MySQL vs SQLite | Which Database Should You Choose?|database-comparison|
A comprehensive comparison...
```

Khi CSV reader đến dòng đó, nó thấy năm cột thay vì ba. Title field trở thành PostgreSQL vs MySQL vs SQLite, slug trở thành Which Database Should You Choose?, và meta description parser bị nhầm lẫn.

Ảnh được tạo cho bài đó có title kiểu như “Hero image for ‘PostgreSQL vs MySQL vs SQLite’” — cắt ngắn title giữa chừng, bỏ câu hỏi thực sự, và tạo ra ảnh database chung chung thay vì ảnh tập trung vào so sánh.

Chúng tôi chỉ phát hiện ra trong lúc review. 23 bài có prompt bị sai.

Cách Sửa: Tự Động Phát Hiện Và Chuyển Sang 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}')"

```

Sau cách sửa này, chúng tôi chạy lại 23 bài bị ảnh hưởng với prompt đúng.

Checkpoint và Resume: Không Bao Giờ Bắt Đầu Lại Từ Đầu

Nguyên Tắc

Bốn tiếng xử lý batch không bao giờ nên bị bỏ đi. Mọi batch job chạy lâu đều cần checkpointing. Thiết kế đơn giản:

- Ghi checkpoint sau mỗi thao tác thành công
- Khi restart, load checkpoint và bỏ qua các item đã hoàn thành
- Theo dõi failure riêng — chúng có thể cần review thủ công hoặc 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")

```

Resume Sau Khi Thất Bại

```

# 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\"]}')
"
```

Thí Nghiệm Logo: Khi AI Không Phải Công Cụ Phù Hợp

Đề Xuất

Chúng tôi cần logo cho blog. Chúng tôi có Gemini. Tại sao phải trả tiền cho designer?

Câu trả lời, sau 47 lần thử tạo logo: vì thiết kế logo chuyên nghiệp đòi hỏi vector graphic, chiến lược thương hiệu, và tư duy design system mà các AI image model hiện tại không cung cấp.

Đây là những gì chúng tôi thực sự nhận được:

Lần thử 1-10: Logo tech chung chung. Vòng tròn xanh, mạch điện trừu tượng, đủ thứ quen thuộc.

Lần thử 11-20: Gần hơn với mô tả của chúng tôi, nhưng styling không nhất quán. Mỗi ảnh trông như đến từ agency thiết kế khác nhau.

Lần thử 21-30: Chúng tôi bắt đầu thêm style reference. Thẩm mỹ tốt hơn, nhưng text rendering bị hỏng — các chữ cái hợp nhau, font là tưởng tượng, các từ là vô nghĩa.

Lần thử 31-40: Tập trung vào wordmark với hướng dẫn font cụ thể. Gemini không thể render font cụ thể một cách đáng tin cậy. Các hình chữ cái sai.

Lần thử 41-47: Chỉ abstract mark, không có text. Những cái này thực sự có thể dùng được làm brand element, nhưng không phải logo — chúng mang tính trang trí.

```
# 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",
]

```

Phán quyết: AI image generation xuất sắc cho blog hero image, social graphic, và illustration. Nó không thay thế được thiết kế logo, vì:

1. Vector output không có sẵn (chỉ raster — không scale được)
2. Text rendering không đáng tin cậy cho typography cụ thể
3. Tính nhất quán qua các biến thể đòi hỏi iteration thủ công mà designer làm tốt hơn
4. Chiến lược thương hiệu (ý nghĩa màu sắc, khác biệt cạnh tranh) nằm ngoài phạm vi

Chúng tôi thuê designer cho logo. \$800, ba lần revision, giao hàng trong bốn ngày. Xứng đáng.

Rate Limit: Nút Cỗ Chai textembedding-gecko

Tình Huống

Song song với image generation, chúng tôi cũng đang xây dựng semantic search trên nội dung blog. Điều này đòi hỏi tạo embedding cho mỗi bài dùng `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) -> list:
    """
    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

```

Quota mặc định là 5 requests per minute. Với gần 300 bài, đó là khoảng một tiếng tạo embedding — giả sử không có lỗi nào và timing hoàn hảo.

Trong thực tế: 90 phút với error recovery.

Yêu Cầu Tăng Quota

Google Cloud quota UI hiển thị thanh trượt cho `textembedding-gecko`. Mức tối đa self-service là 5 RPM — giống mức mặc định. Thanh trượt không tăng cao hơn được.

```

Current quota: 5 requests/minute
Maximum self-service quota: 5 requests/minute

```

Để vượt quá 5 RPM, bạn cần liên hệ Google Cloud sales hoặc support, giải thích use case, và chờ review thủ công.

```

# 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

```

Trong khi chờ tăng quota (chúng tôi yêu cầu 500 RPM, được cấp 60 RPM sau năm ngày), chúng tôi batch embedding request:

```
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 giảm thời gian embedding của chúng tôi từ 90 phút xuống 20 phút — mà không cần tăng quota.

Veo 3: Video Generation

Thí Nghiệm

Giữa chừng dự án, Veo 3 có sẵn trên Vertex AI. Chúng tôi quyết định test nó cho các clip giải thích ngắn kèm theo một số bài blog.

```
# 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

```

Điều Thực Sự Đã Xảy Ra

Authentication hoạt động tốt. API call đi qua. Nhưng dưới tải, Veo 3 đang trả về 503 error: “The server encountered a high load. Please try again.”

Chúng tôi không dưới tải cao. Chúng tôi đang thực hiện một request mỗi lần. Vấn đề là Veo 3 trong preview bị giới hạn tài nguyên phía Google, không phải phía chúng tôi.

```
[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
```

Bài học: model preview chưa sẵn sàng cho production. Chúng tôi chuyển video generation vào backlog và ship mà không có nó.

POD Skill: Năm Phiên Bản Script

Agent quản lý workflow này đang dùng một skill gọi là “POD” (Publish on Demand). Trong suốt dự án, các script `add_text.py` và `generate_image_vertex.py` bên trong skill đã qua năm phiên bản:

- **v1:** Chức năng cơ bản, credential hardcoded (phát hiện ngay)
- **v2:** Environment variable, pipe delimiter, không có error handling
- **v3:** TAB delimiter fix, retry logic cơ bản, không có checkpointing
- **v4:** Checkpointing đầy đủ, quota rotation, batched embedding
- **v5:** Tất cả những thứ trên cộng thêm tích hợp Veo 3 (một phần, tắt trong production)

Sự tiến hóa của các script này đại diện cho đường học thực tế của việc đưa AI generation vào production. Bạn bắt đầu đơn giản và chỉ thêm độ phức tạp khi thất bại đòi hỏi.

```
# Skill directory structure after v5
~/agent/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
```

Bài Học Rút Ra

Test trên 10 bài trước khi chạy trên hàng trăm. Lần chạm quota ở bài 147 có thể tránh được. Chạy 10 bài, xác minh chất lượng output, kiểm tra tốc độ tiêu thụ quota, và ước tính thời gian hoàn thành. Sau đó mới scale.

Checkpointing không phải tùy chọn cho batch job. Bất kỳ batch nào mất hơn 10 phút đều cần checkpoint. Chi phí triển khai là một tiếng. Chi phí không triển khai có thể là nhiều tiếng làm lại.

Delimiter quan trọng. Nếu dữ liệu của bạn chứa ký tự delimiter (và nội dung do người dùng tạo thường có), hãy dùng delimiter an toàn. TAB hoạt động với hầu hết nội dung. Nếu bạn cần tương thích CSV, dùng quoting đúng cách.

Biết khi nào AI không phải công cụ phù hợp. Blog hero image: tuyệt vời. Social graphic: tuyệt vời. Logo: không. Ảnh nhiều text với typography cụ thể: không. Video trong preview: chưa. Khả năng là thực; ranh giới cũng thực.

Tăng quota đòi hỏi thời gian chuẩn bị. Nếu bạn đang lên kế hoạch một batch job lớn, hãy yêu cầu tăng quota một tuần trước khi cần. Giới hạn self-service rất thận trọng. Tăng thủ công mất vài ngày.

Batch API call khi có thể. `textembedding-gecko` hỗ trợ 5 text mỗi call. Một call mỗi text tốn quota gấp 5 lần so với batching. Đọc tài liệu API về batch limit trước khi viết generation loop.

Gần 300 bài được ship với AI-generated image. Chúng trông tốt — không phải không thể phân biệt được với nhiếp ảnh commissioned, nhưng đủ chuyên nghiệp cho một blog kỹ thuật. Tổng chi phí: ~\$120 phí API, ba ngày làm việc của developer, và kiến thức được để cập trong chương này.

Tiếp theo: Chương 13 — Knowledge Management: xây dựng RAG system với Cognee, Neo4j, và 400 cuốn sách DevOps.

Chương 13: Quản Lý Tri Thức: RAG, Đồ Thị, và Bộ Nhớ

“Chúng tôi nhập 400 cuốn sách DevOps, 877MB file PDF, vào một hệ thống bộ nhớ kết hợp đồ thị-vector. Rồi các agent bắt đầu nhâm lẫn tài liệu Kubernetes với phân tích SEO của khách hàng thiết kế nội thất. Hóa ra, việc cô lập dataset không tự động xảy ra.”

Tại Sao RAG Đơn Thuần Là Chưa Đủ

Mọi hệ thống AI agent nghiêm túc đều sớm hay muộn va phải cùng một bức tường: agent không nhớ những gì nó đã làm tuần trước. Bạn có thể nhồi context vào prompt, nhưng context window sẽ đầy. Bạn có thể dùng lịch sử hội thoại, nhưng lịch sử tốn kém và cuối cùng bị cắt bớt. Bạn có thể ghi chú ra file, nhưng file không có tính năng tìm kiếm ngữ nghĩa.

Retrieval-Augmented Generation (RAG) giải quyết bài toán hồi tưởng bằng cách cung cấp cho agent một bộ nhớ ngoài có khả năng tìm kiếm. Nhưng RAG thuần vector — tách đoạn văn bản, nhúng, lưu trữ, truy xuất bằng cosine similarity — cũng có những hạn chế riêng:

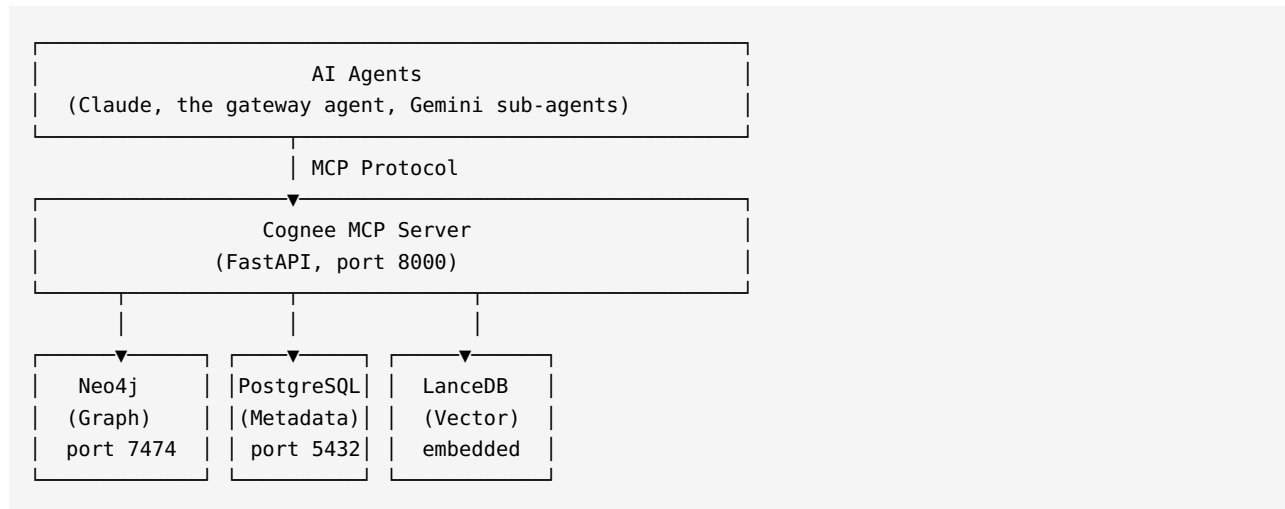
- Nó truy xuất các đoạn có ngữ nghĩa tương tự, không phải các đoạn có quan hệ nhân quả
- Nó không có khái niệm “hai sự kiện này có liên quan” trừ khi chúng nằm liền kề trong văn bản nguồn
- Nó không thể trả lời “chúng ta đã quyết định gì hôm thứ Ba” — nó chỉ tìm được các tài liệu khớp với “quyết định thứ ba”
- Nó xem tất cả tri thức là văn bản phẳng có cấu trúc đồng đều

RAG tăng cường đồ thị bổ sung một lớp quan hệ có cấu trúc. Bạn có thể hỏi: “Những service nào phụ thuộc vào database này?” và nhận câu trả lời được xây dựng từ việc duyệt các kết nối, không chỉ khớp từ khóa.

Cognee kết hợp cả hai phương pháp. Chương này đề cập đến những gì chúng tôi thực sự xây dựng, những thất bại trên đường đi, và hệ thống trông như thế nào khi vận hành thực tế.

Tổng Quan Kiến Trúc

Bộ Công Nghệ



Neo4j lưu trữ đồ thị tri thức: các thực thể (khái niệm, service, công nghệ) và các mối quan hệ của chúng (phụ thuộc-vào, liên-quan-đến, được-dùng-bởi, định-nghĩa-trong).

PostgreSQL lưu trữ metadata tài liệu, thành viên dataset, lịch sử nhập liệu, và trạng thái hoạt động của pipeline Cognee.

LanceDB lưu trữ các vector embedding cho tìm kiếm similarity ngữ nghĩa. Nó chạy nhúng trong tiến trình Cognee — không cần service riêng biệt.

Cognee MCP Server hiển thị hệ thống bộ nhớ cho các AI agent thông qua Model Context Protocol. Các agent gọi `cognify` để lưu trữ tri thức và `search` để truy xuất.

Tại Sao Server-3?

Chúng tôi chọn Server-3 cho việc triển khai toàn bộ stack. Các yếu tố quyết định:

- **RAM:** 9.3GB trống sau các service hiện tại — đủ cho Neo4j (2GB), PostgreSQL (1GB), và tiến trình Cognee (1-2GB)
- **Disk:** 200GB khả dụng — cần thiết cho bộ sưu tập sách 877MB cộng với lưu trữ index
- **Mạng:** Chỉ định tuyến VPN nội bộ — không cần tiếp xúc công khai
- **Các service hiện có:** Không có xung đột port trên 7474, 7687, hoặc 8000

```
# Kiểm tra tài nguyên trước khi triển khai
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)'
# (không có output – các port khả dụng)
```

Cài Đặt Cognee

Cài Đặt và Cấu Hình

```
#!/bin/bash
# install-cognee-stack.sh

set -euo pipefail

# Tạo thư mục ứng dụng
sudo mkdir -p /opt/cognee
sudo chown $USER:$USER /opt/cognee
cd /opt/cognee

# Môi trường ảo Python
python3 -m venv .venv
source .venv/bin/activate

# Cài Cognee với tất cả backend
pip install cognee[neo4j,postgres,lancedb]

# Cài dependencies cho MCP server
pip install "mcp[server]" fastapi uvicorn

# Kiểm tra cài đặt
python3 -c "import cognee; print(cognee.__version__)"

# /opt/cognee/config.py
# Cấu hình Cognee

import os
from cognee.api.v1.config import get_cognee_config

def configure_cognee():
    config = get_cognee_config()

    # Lưu trữ vector – LanceDB (nhúng, không cần service riêng)
    config.set("vector_db_provider", "lancedb")
    config.set("vector_db_path", "/opt/cognee/data/lancedb")

    # Lưu trữ đồ thị – 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"])
```

```

# Lưu trữ quan hệ – 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 để trích xuất tri thức
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"])

# Mô hình embedding
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 cho Neo4j

```

# /opt/cognee/docker-compose.yml

services:
  neo4j:
    image: neo4j:5.18-community
    restart: unless-stopped
    environment:
      NE04J_AUTH: "neo4j/${NE04J_PASSWORD}"
      NE04J_PLUGINS: '["apoc"]'
      NE04J_dbms_memory_heap_initial_size: "512m"
      NE04J_dbms_memory_heap_max_size: "2g"
      NE04J_dbms_memory_pagecache_size: "512m"
    volumes:
      - neo4j-data:/data
      - neo4j-logs:/logs
    ports:
      # Chỉ bind vào interface VPN
      - "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:
      - NE04J_URL=bolt://neo4j:7687
      - NE04J_USER=neo4j
      - NE04J_PASSWORD=${NE04J_PASSWORD}
      - POSTGRES_HOST=10.8.0.12
      - POSTGRES_DB=cognee
      - POSTGRES_USER=${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

```

Tổ Chức Dataset

Bốn Dataset

Chúng tôi tổ chức tri thức thành bốn dataset được đặt tên, mỗi cái có mục đích riêng biệt:

```
# dataset-config.py
```

```

DATASETS = {
  "seo-interior-design-client": {
    "description": "Phân tích SEO cho khách hàng thiết kế nội thất",
    "sources": ["keyword research", "competitor analysis", "content briefs"],
    "agents": ["seo-specialist"],
    "retention": "project-lifetime",
  },
  "agent-knowledge-devops-official": {
    "description": "Tài liệu chính thức: Terraform, Kubernetes, AWS, GCP",
    "sources": ["docs.terraform.io", "kubernetes.io", "docs.aws.amazon.com"],
    "agents": ["gateway-agent", "claude-code"],
    "retention": "permanent",
    "update_frequency": "monthly",
  },
  "agent-knowledge-devops-community": {
    "description": "Tri thức cộng đồng: blog, kinh nghiệm thực tế",
    "sources": ["medium", "dev.to", "personal blogs", "conference talks"],
    "agents": ["gateway-agent", "claude-code"],
    "retention": "permanent",
    "update_frequency": "weekly",
  },
  "agent-memory": {
    "description": "Bộ nhớ dài hạn của agent: quyết định, kết quả, sở thích",
    "sources": ["agent-generated", "conversation summaries"],
    "agents": ["gateway-agent", "claude-code"],
    "retention": "permanent",
    "update_frequency": "continuous",
  },
}

```

Tại Sao Cô Lập Dataset Quan Trọng

Chúng tôi học được tầm quan trọng của việc cô lập dataset theo cách khó quên. Trước khi áp dụng ranh giới dataset đúng đắn, cả hai agent đều dùng cùng một dataset mặc định.

Phân tích SEO cho khách hàng thiết kế nội thất — nghiên cứu từ khóa, phân tích đối thủ, nội dung về vải rèm cửa và màu sơn — bị trộn lẫn vào cùng không gian vector với tài liệu Kubernetes và các module Terraform.

Khi infrastructure agent tìm kiếm “chiến lược triển khai”, nó truy xuất hỗn hợp gồm: - Tài liệu về chiến lược triển khai Kubernetes (đúng) - Bài đăng về chiến lược phát triển kinh doanh của nhà thiết kế nội thất (sai) - Brief nội dung về trang trí các phong cách nhà khác nhau (rất sai)

Kết quả tìm kiếm bị pha loãng. Tệ hơn, LLM đôi khi kết hợp context thiết kế nội thất vào các câu trả lời về infrastructure, tạo ra những câu trả lời sai một cách tinh tế nhưng tự tin.

```
# cognee-dataset-manager.py
# Cô lập dataset đúng đắn cho các thiết lập agent đa thuê

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:
    """
    Thêm nội dung vào một dataset được đặt tên cụ thể.
    Các agent chỉ tìm kiếm trong các dataset được gán cho họ.
    """
    # Đặt context dataset hiện hoạt
    await cognee.prune.prune_system(metadata=True) # Reset context dataset

    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]:
    """
    Tìm kiếm chỉ trong các dataset được chỉ định.
    Không bao giờ tìm kiếm ngoài phạm vi dataset được chỉ định.
    """
    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)

    # Xếp hạng lại theo điểm trên tất cả kết quả
    all_results.sort(key=lambda x: x.get("score", 0), reverse=True)
    return all_results[:limit]

```

Nhập 400 Cuốn Sách DevOps

Pipeline Nhập Liệu

Chúng tôi có bộ sưu tập 838 file PDF, tổng cộng 877MB — sách giáo khoa, sách O'Reilly, kỹ yếu hội nghị, và tài liệu tham khảo về Kubernetes, Terraform, AWS, observability, thực hành SRE, và bảo mật.

```

#!/usr/bin/env python3
# import-devops-books.py
# Nhập bộ sưu tập sách DevOps vào đồ thị tri thức Cognee

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 = "agent-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:
    """Nhập một file PDF vào Cognee. Trả về True nếu thành công."""
    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:
                # Tài liệu đã tồn tại – coi là thành công
                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" Giới hạn tốc độ. Chờ {wait}s (lần thử {attempt+1}/{retries})")
                await asyncio.sleep(wait)
            else:
                print(f" Lỗi (lần thử {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"Tim thấy {total} file PDF ({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"Tiếp tục: {len(completed_set)}/{total} đã nhập")

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)

    # Làm chậm để tránh làm quá tải pipeline
    await asyncio.sleep(3)

print(f"\nNhập xong: {len(cp['completed'])} thành công, {len(cp['failed'])} thất bại")
if cp["failed"]:
    print("Các file thất bại:")
    for item in cp["failed"]:
        print(f"  {item['file']}")

if __name__ == "__main__":
    asyncio.run(main())

```

Câu Chuyện Thực Tế: Con Bão 409 Conflict

Khoảng file thứ 200 trong quá trình nhập, chúng tôi bắt đầu thấy một loạt lỗi 409 Conflict.

```

[203/838] kubernetes-in-action-2nd-edition.pdf
Lỗi (lần thứ 1/3): 409 Conflict - Tài liệu 'kubernetes-in-action-2nd-edition.pdf'
đã tồn tại trong dataset
Lỗi (lần thứ 2/3): 409 Conflict
Lỗi (lần thứ 3/3): 409 Conflict
THẤT BẠI: kubernetes-in-action-2nd-edition.pdf

```

Script đang thử lại các lỗi 409 như thế chúng là lỗi tạm thời. Lỗi 409 Conflict có nghĩa là tài liệu đã tồn tại — thử lại sẽ luôn tạo ra cùng một lỗi.

Cách sửa rất đơn giản: coi 409 là thành công (nhập liệu idempotent).

```
# Cách sửa – trong import_pdf()
if "409" in error_str or "Conflict" in error_str:
    # Tài liệu đã tồn tại – điều này ổn thôi
    # Cognee dùng hash nội dung để loại trùng lặp
    log_event({"event": "already_exists", "file": pdf_path.name})
    return True # Không phải lỗi
```

Sau khi sửa, chúng tôi nhận ra các lỗi 409 này đến từ một lần thử nhập trước đó bị bỏ dở. Checkpoint từ lần chạy đó đã bị mất khi server khởi động lại. Việc xử lý idempotent có nghĩa là chạy lại là an toàn.

Bài học: Thiết kế pipeline nhập liệu idempotent từ đầu. “Đã tồn tại” không phải lỗi — đó là xác nhận.

Câu Chuyện Thực Tế: Giới Hạn Tốc Độ Vertex AI Làm Đình Trệ Mọi Thứ

Giữa chừng nhập liệu, mọi thứ đình lại. Không phải crash — đình lại. Script vẫn chạy nhưng tiến trình đã chậm xuống còn một file mỗi 3-4 phút.

```
# Chẩn đoán xem có chuyện gì
tail -f /opt/cognee/import-log.jsonl | grep -E '(rate_limit|429|ResourceExhausted)'
```

Pipeline cognify của Cognee — vốn trích xuất thực thể và quan hệ từ văn bản — đang gọi embedding API cho mỗi chunk của mỗi tài liệu. Với 838 file, mỗi file trung bình hơn 100 trang, chúng tôi đang tạo ra hàng nghìn yêu cầu embedding.

Chúng tôi đã chạm đến quota `textembedding-gecko`. Giới hạn 5 RPM tương tự như ở Chương 12.

Cognee không có cách kiểm soát trực tiếp giới hạn tốc độ trong API công khai của nó. Các lệnh gọi embedding xảy ra bên trong pipeline cognify. Các lựa chọn của chúng tôi:

1. Chờ tăng quota (vài ngày)
2. Chuyển mô hình embedding sang loại có quota cao hơn
3. Làm chậm pipeline nhập liệu (giảm tốc độ nhân tạo để ở dưới quota)

Chúng tôi chọn phương án 3 trước, rồi đến phương án 2.

```
# Nhập với điều tiết – thêm delay giữa các file
# Tính toán: 5 RPM, tài liệu trung bình tạo ~20 lệnh gọi embedding
# 20 lệnh / 5 RPM = tối thiểu 4 phút mỗi tài liệu
```

```

# Thêm buffer: chờ 5 phút giữa các tài liệu lớn

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)

    # Điều tiết dựa trên kích thước file (lớn hơn = nhiều chunk hơn = nhiều lệnh gọi embedding hơn)
    if size_mb > 10:
        wait = 300 # 5 phút cho file lớn
    elif size_mb > 5:
        wait = 120 # 2 phút cho file trung bình
    else:
        wait = 30 # 30s cho file nhỏ

    print(f" Điều tiết {wait}s (kích thước file: {size_mb:.1f}MB)")
    await asyncio.sleep(wait)

    return success

```

Việc nhập hoàn thành sau bốn ngày thay vì bốn giờ. Không lý tưởng, nhưng đã hoàn thành.

MCP Server: Tri Thức Được Truy Cập Bởi Agent

Cấu Hình MCP

MCP (Model Context Protocol) server hiển thị khả năng tìm kiếm và lưu trữ của Cognee cho các AI agent dưới dạng các công cụ có thể gọi được.

```

# /opt/cognee/mcp-server.py
# MCP server hiển thị Cognee như các công cụ agent

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=(
    "Lưu thông tin vào bộ nhớ dài hạn của agent. "
    "Dùng cho các quyết định, phát hiện, cấu hình, và sự kiện "
    "cần được duy trì qua các phiên làm việc."
),
inputSchema={
    "type": "object",
    "properties": {
        "content": {
            "type": "string",
            "description": "Thông tin cần lưu trữ"
        },
        "dataset": {
            "type": "string",
            "description": "Tên dataset (ví dụ: 'agent-memory')",
            "default": "agent-memory"
        },
        "tags": {
            "type": "array",
            "items": {"type": "string"},
            "description": "Các tag tùy chọn để phân loại"
        }
    },
    "required": ["content"]
},
),
Tool(
    name="search_memory",
    description=(
        "Tìm kiếm bộ nhớ agent để lấy thông tin liên quan. "
        "Trả về nội dung có ngữ nghĩa tương tự và các node đồ thị liên quan."
    ),
    inputSchema={
        "type": "object",
        "properties": {
            "query": {
                "type": "string",
                "description": "Nội dung cần tìm kiếm"
            },
            "datasets": {
                "type": "array",
                "items": {"type": "string"},
                "description": "Các dataset cần tìm kiếm (bỏ qua để dùng dataset mặc
định)"
            },
            "limit": {
                "type": "integer",
                "default": 10,
                "description": "Số kết quả tối đa trả về"
            }
        },
        "required": ["query"]
    },
),
Tool(
    name="get_graph_neighbors",
    description=(
        "Lấy các khái niệm liên quan từ đồ thị tri thức. "
        "Dùng để khám phá các mối quan hệ giữa các thực thể."
    ),
    inputSchema={

```

```

        "type": "object",
        "properties": {
            "entity": {
                "type": "string",
                "description": "Tên thực thể cần khám phá"
            },
            "depth": {
                "type": "integer",
                "default": 2,
                "description": "Độ sâu duyệt đồ thị"
            }
        },
        "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", "agent-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 content
                    })
                )]
            )

        elif request.name == "search_memory":
            query = request.arguments.get("query", "")
            datasets = request.arguments.get("datasets", ["agent-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)

    # Truy vấn Neo4j trực tiếp để duyệt đồ thị
    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())

```

Cấu Hình Claude Code MCP

```

// ~/.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: Chọn Công Cụ Thu Thập Web

Chúng tôi đánh giá cả hai công cụ để diễn dataset tri thức cộng đồng với các bài đăng blog, tài liệu, và các thread diễn đàn.

Firecrawl

Firecrawl là một dịch vụ thu thập toàn site. Bạn cung cấp cho nó một domain và nó theo dõi các link, tuân thủ robots.txt, xử lý rendering JavaScript, và trả về markdown sạch.

Điểm mạnh: - Thu thập toàn site với hỗ trợ sitemap - Thu thập sâu (theo tất cả link đến độ sâu cấu hình) - Xử lý các site nặng JavaScript - Trả về markdown có cấu trúc, không phải HTML thô

Điểm yếu: - Hạ tầng nặng: cần Node.js, Redis, và bản thân dịch vụ Firecrawl - Triển khai tự host không đơn giản - Tốn nhiều API call hơn cho việc trích xuất một trang được nhắm mục tiêu

Triển khai:

```

# 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:

# 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]:
    """Thu thập toàn bộ site và trả về markdown sạch cho mỗi trang."""
    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"Công việc thu thập đã bắt đầu: {job_id}")

    # Kiểm tra hoàn thành định kỳ
    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"Thu thập thất bại: {status_data.get('error')}")

        print(f"Đang thu thập... {status_data.get('completed', 0)}/{status_data.get('total',
'?'}})")

        time.sleep(5)

```

Crawl4AI

Crawl4AI là một thư viện Python nhẹ cho việc trích xuất một trang với các schema đầu ra có cấu trúc.

Điểm mạnh: - Python thuần túy — không cần Node.js, không cần Redis, dependencies tối thiểu - Xuất sắc cho việc trích xuất có mục tiêu với các schema - Nhanh cho các trường hợp dùng một trang hoặc nhỏ - Schema trích xuất có thể cấu hình để lấy dữ liệu có cấu trúc

Điểm yếu: - Không có tính năng thu thập toàn site tích hợp (bạn tự quản lý việc theo link) - Rendering JavaScript kém tinh vi hơn Firecrawl

```
# crawl4ai-ingestion.py

import asyncio
from crawl4ai import AsyncWebCrawler, BrowserConfig, CrawlerRunConfig
from crawl4ai.extraction_strategy import JsonCssExtractionStrategy
import json

# Schema để trích xuất cấu trúc bài đăng blog
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:
    """Trích xuất nội dung có cấu trúc từ một bài đăng blog đơn lẻ."""
    browser_config = BrowserConfig(headless=True, verbose=False)
    run_config = CrawlerRunConfig(
        extraction_strategy=JsonCssExtractionStrategy(BLOG_POST_SCHEMA),
        cache_mode="enabled", # Cache phản hồi để chạy lại
    )

    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] # Bài viết đầu tiên khớp
            except json.JSONDecodeError:
                # Dự phòng sang nội dung markdown
                return {"content": result.markdown, "url": url}

    return None
```

```

async def batch_extract_posts(urls: list[str], concurrency: int = 3) -> list[dict]:
    """Trích xuất nhiều bài đăng blog với concurrency có kiểm soát."""
    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) # Delay lịch sự

    await asyncio.gather(*[extract_with_limit(url) for url in urls])
    return results

```

Ma Trận Quyết Định

Yếu tố	Firecrawl	Crawl4AI
Thu thập toàn site	Có	Thủ công
Overhead hạ tầng	Cao (Node, Redis)	Tối thiểu
Rendering JavaScript	Xuất sắc	Tốt
Trích xuất có cấu trúc	Cơ bản	Xuất sắc (schema)
Thời gian cài đặt	2-4 giờ	15 phút
Phù hợp nhất với	Toàn site	Trang được nhắm mục tiêu

Quyết định của chúng tôi: Firecrawl cho các site tài liệu chính thức (thu thập mọi thứ), Crawl4AI cho việc trích xuất có mục tiêu các bài đăng blog và bài viết cụ thể. Chúng bổ sung cho nhau.

Bộ Nhớ Thần Kinh: Đồ Thị Trông Như Thế Nào

Sau khi hoàn thành toàn bộ việc nhập — 400 cuốn sách, các lần chạy Firecrawl trên các site tài liệu, và các lần trích xuất Crawl4AI có mục tiêu — chúng tôi chạy `cognee.get_graph_statistics()` để xem chúng tôi đã xây dựng được gì.

```

# 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"Loại node: {stats['node_types']}")
print(f"Loại edge: {stats['edge_types']}")

# Lấy mẫu một số node ngẫu nhiên
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())

```

Kết quả sau khi cài đặt hoàn chỉnh:

```

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 node và 78.431 edge. Đồ thị đại diện cho cấu trúc khái niệm của tri thức vận hành của chúng tôi — các khái niệm được kết nối với các tài liệu định nghĩa chúng, các công nghệ được kết nối với các quy trình sử dụng chúng, các quy trình được kết nối với các sự cố thúc đẩy chúng.

Khi một agent tìm kiếm “Kubernetes pod scheduling”, nó không chỉ nhận lại văn bản tương tự, mà còn các khái niệm liên quan: node affinity, resource requests, pod disruption budgets, và các trang cụ thể trong các sách chúng tôi đã nhập bao gồm từng khái niệm. Duyệt đồ thị biến tìm kiếm từ khóa thành một phiên điều hướng tri thức.

Bài Học Rút Ra

Cô lập dataset là kiến trúc, không phải cấu hình. Đặt tất cả tri thức vào một namespace tương đương với việc đặt tất cả code vào một file. Bạn sẽ sớm hối tiếc điều đó. Thiết kế cấu trúc dataset trước khi bắt đầu nhập.

Import idempotent ngay từ ngày đầu. Coi “đã tồn tại” là thành công. Lỗi mạng, khởi động lại, và hết quota sẽ làm gián đoạn các công việc nhập dài. Pipeline của bạn cần tiếp tục từ nơi nó dừng lại mà không xử lý lại các mục đã hoàn thành.

Giới hạn tốc độ là trên, không phải mục tiêu. Giới hạn 5 RPM của textembedding-gecko không phải để vận hành ở đó — mà là để vận hành ở mức thấp hơn nhiều, với dư địa để thử lại. Xây dựng điều tiết giữ bạn ở 80% giới hạn.

Bộ nhớ đồ thị và bộ nhớ vector phục vụ các truy vấn khác nhau. Vector: “tìm cho tôi thứ gì đó tương tự với cái này.” Đồ thị: “còn những gì khác kết nối với khái niệm này?” Bạn cần cả hai. RAG thuần vector trả lời câu hỏi về nội dung. RAG tăng cường đồ thị trả lời câu hỏi về mối quan hệ.

Crawl4AI và Firecrawl bổ sung cho nhau. Chọn dựa trên nguồn. Site tài liệu chất lượng cao với cấu trúc sâu: Firecrawl. Trích xuất có mục tiêu với schema đã biết: Crawl4AI. Đừng chọn một cái và ép nó cho mọi thứ.

Theo dõi pipeline, không chỉ đầu ra. Chúng tôi chỉ phát hiện ra sự đình trệ giới hạn tốc độ bằng cách xem log. Thêm metrics: số lệnh gọi embedding mỗi phút, số tài liệu trong hàng đợi, tốc độ xử lý, tần suất lỗi. Một pipeline bị đình trệ mà không báo lỗi còn tệ hơn là một pipeline thất bại to tiếng.

Hệ thống chúng tôi xây dựng không hoàn hảo — câu chuyện về cô lập dataset là bài học cảnh giác về việc bỏ qua các quyết định kiến trúc dưới áp lực thời gian. Nhưng với 6.839 node và 78.431 synapse, các agent có quyền truy cập vào một thập kỷ tri thức DevOps tập thể, có thể truy xuất trong mili giây.

Tiếp theo: Chương 14 — Điều Phối Đa Agent Trong Thực Tế: tmux, hộp thư đến chung, và điều gì xảy ra khi một sub-agent không tìm thấy thư mục không tồn tại.

Chương 14: Điều Phối Đa Agent Trong Thực Tế

“Sub-agent báo cáo thành công. Việc triển khai đã thất bại. Thư mục mà nó đáng lẽ phải tạo file vào không tồn tại. Không ai kiểm tra. Agent cứ thế... tiếp tục.”

Vấn Đề Với Một Agent

Một AI agent đơn lẻ có context window. Một khi window đó đầy, context cũ bị loại bỏ. Với một tác vụ infrastructure phức tạp — chẳng hạn, di chuyển một PostgreSQL cluster trong khi cập nhật cấu hình ứng dụng và xoay vòng credentials — bạn sẽ làm đầy window đó trước khi hoàn thành.

Giải pháp ngây thơ là yêu cầu agent trả lời ngắn gọn hơn, tóm tắt khi làm việc, tiết kiệm hơn. Điều này hoạt động một thời gian. Rồi bạn chạm đến 200K token trong một tác vụ kéo dài nhiều giờ, và agent bắt đầu quên những gì nó đã quyết định hai giờ trước.

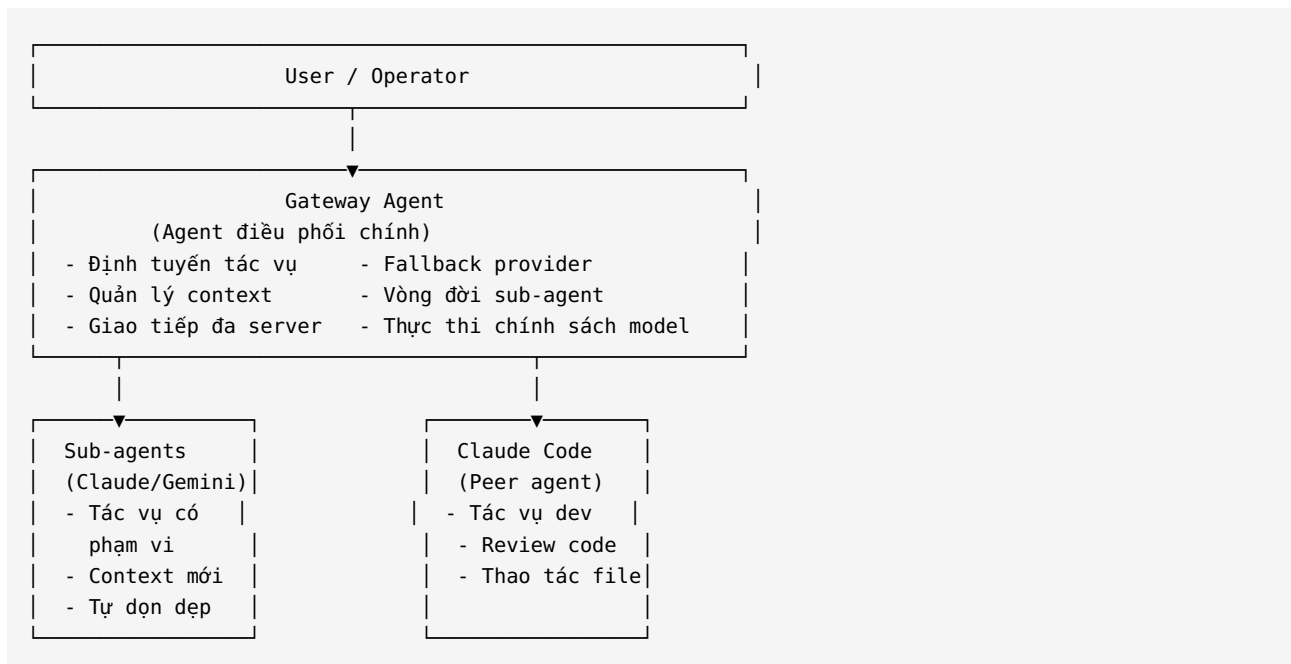
Giải pháp thực sự là điều phối: chia công việc thành các tác vụ có phạm vi, giao mỗi tác vụ cho một agent mới với context window đầy, và điều phối kết quả. Đây là điều phối đa agent, và nó lộn xộn hơn trong thực tế so với nghe có vẻ trong lý thuyết.

Chương này đề cập đến những gì chúng tôi thực sự xây dựng: kiến trúc gateway agent, kênh giao tiếp dựa trên tmux, các chế độ thất bại mà chúng tôi không lường trước, và các mẫu vận hành nổi lên từ việc chạy hệ thống này trong môi trường thực tế.

Kiến Trúc: Mẫu Gateway Agent

Thiết Kế Cốt Lõi

Gateway agent là agent điều phối chính của chúng tôi — luôn chạy, luôn có thể tiếp cận. Nó sở hữu hàng đợi tác vụ, tạo ra các sub-agent cho công việc nặng, và điều phối kết quả. Claude Code là một peer agent với khả năng bổ sung: phân tích code sâu hơn, các thao tác hệ thống file, và truy cập trực tiếp vào môi trường phát triển.



Giao Tiếp Qua Shared Workspace

Gateway agent và coding agent giao tiếp thông qua một shared workspace — một thư mục cả hai agent đều có thể đọc và ghi. Đây là cơ chế truyền tin đơn giản, đáng tin cậy không cần bất kỳ giao thức thời gian thực nào.

```
~/ .agent-gateway/workspace/
├── inbox/
│   ├── from-gateway.md      # Tin nhắn ĐẾN Claude Code
│   └── from-claudecode.md   # Tin nhắn TỪ Claude Code
├── proposals/              # Đề xuất hợp tác
├── research/                # Kết quả nghiên cứu
├── MEMORY.md                # Bộ nhớ bền vững chung
└── memory/
    └── YYYY-MM-DD.md        # Nhật ký vận hành hàng ngày
```

```
# Gateway agent gửi tác vụ đến Claude Code
cat >> ~/ .agent-gateway/workspace/inbox/from-gateway.md << 'EOF'

## Tác vụ: Review Script Migration PostgreSQL
```

Timestamp: 2026-03-01T14:32:00Z

Độ ưu tiên: CAO

Vui lòng review /opt/migrations/pg-migration-v2.sql về:

1. Rủi ro toàn vẹn dữ liệu
2. Vấn đề tranh chấp khóa
3. An toàn rollback

Trả lời trong from-claudecode.md khi hoàn thành.

EOF

Thông báo cho Claude Code qua system event

```
gateway system event --text "Tác vụ đang chờ trong inbox/from-gateway.md – review migration PostgreSQL"
```

Claude Code kiểm tra hộp thư đến (nên xảy ra khi bắt đầu phiên và sau khi nhàn rỗi)

```
cat ~/.agent-gateway/workspace/inbox/from-gateway.md 2>/dev/null
```

Claude Code trả lời sau khi hoàn thành review

```
cat >> ~/.agent-gateway/workspace/inbox/from-claudecode.md << 'EOF'
```

Re: Review Script Migration PostgreSQL

Timestamp: 2026-03-01T14:51:00Z

Trạng thái: HOÀN THÀNH

Kết quả review:

1. Dòng 47: ALTER TABLE với toàn bộ bảng rewrite trên 50M row – dự kiến khóa 20-40 phút
2. Không có transaction boundary rõ ràng – nếu script thất bại giữa chừng, trạng thái một phần
3. Thiếu câu lệnh ROLLBACK cho các thay đổi FK constraint

Khuyến nghị: Thêm BEGIN/COMMIT rõ ràng, thêm kiểm tra LOCK TABLE với NOWAIT trước.

Review chi tiết: /tmp/pg-migration-review.md

EOF

```
gateway system event --text "Review migration hoàn thành. Tìm thấy 3 vấn đề. Chi tiết trong inbox/from-claudecode.md"
```

Giao Tiếp Dựa Trên Tmux: Gửi Tổ Hợp Phím Giữa Các Phiên

Phương pháp shared workspace hoạt động cho các tin nhắn bất đồng bộ.

Nhưng đôi khi gateway agent cần tiêm một tác vụ vào phiên Claude Code đang hoạt động ngay lập tức — không phải “kiểm tra hộp thư đến khi tiện”, mà là “đây là tác vụ, bắt đầu ngay”.

Đây là lúc tmux phát huy tác dụng.

Cách ping-claudecode.sh Hoạt Động

```
#!/bin/bash
# ping-claudecode.sh
# Tiêm tin nhắn vào phiên Claude Code tmux đang hoạt động

set -euo pipefail
```

```

MESSAGE="${1:-Kiểm tra hộp thư đến cho tác vụ mới}"
SESSION_NAME="${CLAUDE_TMUX_SESSION:-claude-code}"
PANE_TARGET="${SESSION_NAME}:0.0"

# Kiểm tra phiên có tồn tại không
if ! tmux has-session -t "$SESSION_NAME" 2>/dev/null; then
    echo "LỖI: Phiên tmux '$SESSION_NAME' không tìm thấy"
    echo "Các phiên đang hoạt động: $(tmux list-sessions 2>/dev/null || echo 'không có')"
    exit 1
fi

# Ghi tin nhắn vào hộp thư đến trước
INBOX_FILE="$HOME/.agent-gateway/workspace/inbox/from-gateway.md"
cat >> "$INBOX_FILE" << EOF

## Tác vụ đến [$(date -u +%Y-%m-%dT%H:%M:%SZ)]
$MESSAGE
EOF

# Tiêm tin nhắn như các tổ hợp phím vào pane Claude Code
# Điều này kích hoạt Claude Code xử lý tin nhắn ngay lập tức
tmux send-keys -t "$PANE_TARGET" "" Enter
sleep 0.3
tmux send-keys -t "$PANE_TARGET" "Tác vụ mới từ gateway: $MESSAGE" Enter

echo "Tin nhắn đã gửi đến $PANE_TARGET"

# Sử dụng từ gateway agent
ping-claudecode.sh "Triển khai script migration đã được review lên staging – đã được phê duyệt"

# Hoặc trực tiếp từ shell
~/agent-gateway/scripts/ping-claudecode.sh "Khẩn cấp: Disk Server-2 ở 94%, cần dọn dẹp"

```

Quản Lý Phiên

Cả hai agent chạy trong các phiên tmux được đặt tên. Quy ước đặt tên quan trọng — đó là cách các script tìm đúng pane.

```

# Khởi động Claude Code trong phiên được đặt tên
tmux new-session -d -s claude-code -x 220 -y 50
tmux send-keys -t claude-code "claude" Enter

# Khởi động gateway agent trong phiên riêng của nó
tmux new-session -d -s gateway-agent -x 220 -y 50
tmux send-keys -t gateway-agent "gateway start" Enter

# Liệt kê các phiên đang hoạt động
tmux list-sessions

# Gắn vào phiên để quan sát
tmux attach -t claude-code

# Gửi tác vụ đến Claude Code mà không cần gắn vào
tmux send-keys -t claude-code:0.0 "Kiểm tra ~/.agent-gateway/workspace/inbox/from-gateway.md" Enter

```

Tạo Sub-Agent: Giao Thức Fast-Track

Khi Nào Tạo Sub-Agent

Không phải mọi tác vụ đều cần sub-agent. Việc tạo thêm overhead: thời gian khởi tạo, thiết lập context, thu thập kết quả. Quy tắc chúng tôi đã thống nhất:

Tạo sub-agent nếu: - Thời gian ước tính tác vụ > 10 giây - Tác vụ cần sử dụng công cụ nhiều (nhiều lần đọc file, gọi mạng, thao tác build) - Tác vụ độc lập và có thể song song hóa - Tác vụ có thể làm đầy context window của agent hiện tại

Xử lý trực tiếp nếu: - Tra cứu hoặc tính toán đơn giản - Một lệnh gọi công cụ duy nhất - Kết quả cần ngay lập tức cho bước tiếp theo

```
# fast-track-decision.py
# Logic quyết định khi nào tạo sub-agent vs xử lý trực tiếp

from enum import Enum

class ExecutionMode(Enum):
    INLINE = "inline"
    SUB_AGENT = "sub_agent"

def decide_execution_mode(task: dict) -> ExecutionMode:
    """
    Quyết định có nên thực thi trực tiếp hay tạo 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)

    # Điều kiện tạo sub-agent
    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

    # Trực tiếp: đơn giản, nhanh, cần kết quả ngay
    return ExecutionMode.INLINE
```

Các Mẫu Tạo Sub-Agent

```
# agent-spawner.py
# Các mẫu tạo sub-agent với quản lý vòng đời đúng đắn

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:
    """
    Tạo một Claude sub-agent cho một tác vụ có phạm vi.
    Luôn dùng --cleanup delete để tránh tích lũy phiên.
    """

    context_parts = []
    if context_files:
        for f in context_files:
            if Path(f).exists():
                context_parts.append(f"File context: {f}")

    full_prompt = f"""
{task_description}

Thư mục làm việc: {working_dir or os.getcwd()}
{chr(10).join(context_parts)}

Khi hoàn thành, ghi tóm tắt ngắn gọn về những gì đã làm vào:
~/agent-gateway/workspace/inbox/from-claudecode.md

Bao gồm: trạng thái (SUCCESS/FAILED), file đã sửa đổi, và bất kỳ vấn đề nào.
"""

    cmd = [
        "claude",
        "--model", model,
        "--print", # Chế độ không tương tác
        "--cleanup", "delete", # LUÔN LUÔN: dọn dẹp phiên sau
        "--max-turns", "50", # Ngăn vòng lặp vô tận
        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 hết thời gian sau {timeout_minutes} phút",
        "elapsed_seconds": timeout_minutes * 60,
        "model": model,
    }

def spawn_gemini_subagent(task_description: str, model: str = "gemini-2.0-flash") -> dict:
    """
    Tạo Gemini sub-agent cho các tác vụ nội dung/nghiên cứu.
    Gemini: Phù hợp cho nội dung, nghiên cứu, phân tích.
    KHÔNG phù hợp cho các thay đổi infrastructure.
    """
    # Cài đặt khác nhau tùy theo cấu hình Gemini CLI
    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,
    }

```

Quy Tắc cleanup: delete

Mỗi lần tạo sub-agent phải bao gồm `--cleanup delete`. Nếu không, các phiên sub-agent sẽ tích lũy. Sau vài ngày điều phối nặng, bạn sẽ có hàng chục phiên mồ côi tiêu tốn bộ nhớ và làm lộn xộn danh sách phiên.

Chúng tôi đã học được điều này sau một cuối tuần xử lý hàng loạt nặng để lại 47 phiên mồ côi.

```

# Dọn dẹp các phiên mồ côi thủ công (sau khi quên --cleanup delete)
claude --list-sessions | grep "sub-agent" | awk '{print $1}' | xargs -I{} claude --delete-session {}

# Hoặc dọn dẹp toàn bộ
claude --list-sessions | grep -v "main\|claude-code" | awk '{print $1}' | xargs -I{} claude --delete-session {}

```

Chính Sách Model: Agent Nào Cho Tác Vụ Nào

Quy Tắc Cứng

Chúng tôi đã đặt một chính sách không thể thương lượng sau một sự cố đau đớn: **các tác vụ infrastructure yêu cầu Claude Opus 4.6. Không có ngoại lệ.**

Sự cố: chúng tôi tạo ra một Gemini sub-agent để sửa một systemd service bị lỗi. Gemini đã thử. Nó chạy `systemctl --user restart service-name`. Service đó là system service, không phải user service. Lệnh thực thi thành công (không có lỗi, user-level service chỉ đơn giản là không tồn tại). Agent báo cáo thành công. Service thực tế vẫn bị down. Chúng tôi không phát hiện ra điều này cho đến khi cảnh báo monitoring kích hoạt 20 phút sau.

```
# model-policy.py
# Thực thi chính sách lựa chọn model cho sub-agent

from enum import Enum

class TaskCategory(Enum):
    INFRASTRUCTURE = "infrastructure" # systemd, mạng, DB, cấu hình server
    CODE_REVIEW = "code_review" # Phân tích và phê bình code
    CODE_WRITING = "code_writing" # Triển khai tính năng, sửa lỗi
    CONTENT = "content" # Blog, tài liệu, bản sao
    RESEARCH = "research" # Tìm kiếm web, phân tích, tóm tắt
    DATA_ANALYSIS = "data_analysis" # Kết quả truy vấn, phân tích log

# Chính sách model – không sửa đổi mà không thảo luận với nhóm
MODEL_POLICY = {
    TaskCategory.INFRASTRUCTURE: {
        "model": "claude-opus-4-6",
        "rationale": "Lỗi infrastructure khó phát hiện và tốn kém để khôi phục",
        "override_allowed": False,
    },
    TaskCategory.CODE_REVIEW: {
        "model": "claude-opus-4-6",
        "rationale": "Chất lượng review code ảnh hưởng trực tiếp đến độ tin cậy production",
        "override_allowed": False,
    },
    TaskCategory.CODE_WRITING: {
        "model": "claude-sonnet-4-6", # Cân bằng chất lượng/chi phí
        "rationale": "Đủ phức tạp để cần model mạnh, không phải lúc nào cũng cần Opus",
        "override_allowed": True,
    },
    TaskCategory.CONTENT: {
        "model": "gemini-2.0-flash",
        "rationale": "Chất lượng nội dung không có chế độ thất bại cứng",
        "override_allowed": True,
    },
    TaskCategory.RESEARCH: {
        "model": "gemini-2.0-flash",
        "rationale": "Nghiên cứu ít rủi ro, Gemini xử lý tốt",
        "override_allowed": True,
    },
    TaskCategory.DATA_ANALYSIS: {
        "model": "gemini-2.0-flash",
        "rationale": "Tác vụ phân tích có cấu trúc, model ít quan trọng hơn 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"CẢNH BÁO: Override model '{requested_model}' bị từ chối cho {category.value}. "
        f"Chính sách yêu cầu: {policy['model']}. Lý do: {policy['rationale']}"
    )

return policy["model"]

```

Tại Sao Gemini Sub-Agent Không Thể Sửa Vấn Đề Hệ Thống

Lỗi systemctl không phải là vấn đề về năng lực của Gemini. Đó là vấn đề về context. Gemini, theo mặc định, không biết:

- Liệu một service là user service hay system service
- File systemd unit thực sự trông như thế nào
- Sự khác biệt giữa service được enabled so với được started
- Sự khác biệt giữa `journalctl -u` và `journalctl --user-unit`

Claude Opus 4.6, được giao cùng tác vụ, đặt câu hỏi làm rõ hoặc kiểm tra cấu hình service trước khi hành động. Không phải Gemini không thể đưa ra câu trả lời đúng — mà là công việc infrastructure đòi hỏi một model biết khi nào nó không biết đủ để tiến hành.

Các Chế Độ Thất Bại Của Sub-Agent

Chế Độ Thất Bại 1: Giới Hạn Context 200K

Sub-agent, giống như agent chính, có giới hạn context. Một sub-agent được giao tác vụ “migrate toàn bộ schema database” sẽ chạm giới hạn trước khi hoàn thành nếu schema lớn và migration phức tạp.

Triệu chứng: sub-agent bắt đầu tạo ra các phản hồi ngắn và ngắn hơn, ngừng sử dụng công cụ, và cuối cùng cắt bớt đầu ra của nó.

Cách sửa: phạm vi tác vụ hẹp hơn. Thay vì “migrate database,” tạo: - Sub-agent A: migrate bảng 1-20 - Sub-agent B: migrate bảng 21-40 - Sub-agent C: cập nhật các foreign key constraint

```

# task-splitter.py
# Chia các tác vụ lớn thành các chunk kích thước phù hợp với sub-agent

def split_schema_migration(tables: list[str], chunk_size: int = 20) -> list[dict]:
    """Chia một migration schema lớn thành các sub-tác vụ có thể quản lý."""

```

```

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 bảng {i+1} đến {min(i+chunk_size, len(tables))}: {'',
'.join(chunk[:3])}...",
        "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:
    """
    Ước tính thô số token context một tác vụ sẽ dùng.
    Trả về số token ước tính.
    """
    # Ước tính thô: 1 token mỗi 4 ký tự
    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

    # Thêm buffer cho overhead lệnh gọi công cụ
    overhead = 50_000 # Overhead điển hình cho tác vụ phức tạp

    return desc_tokens + file_tokens + overhead

def should_split_task(task: dict, limit: int = 150_000) -> bool:
    """Xác định liệu tác vụ có cần chia nhỏ trước khi tạo sub-agent."""
    estimated = estimate_context_usage(
        task.get("description", ""),
        task.get("context_files", []),
    )
    if estimated > limit:
        print(f"Ước tính context tác vụ {estimated:,} token vượt giới hạn {limit:,}. Đang chia nhỏ.")
        return True
    return False

```

Chế Độ Thất Bại 2: Cấp Độ Service Sai (User vs System Systemd)

Lỗi systemctl từ phần chính sách model là một loại thất bại đáng ghi lại rõ ràng. Các AI agent không phải lúc nào cũng biết sự khác biệt giữa:

```

systemctl status service-name # System service (cần sudo)
systemctl --user status service-name # User service (context người dùng hiện tại)

```

Một system service chạy dưới `root` hoặc một service user chuyên dụng sẽ không xuất hiện trong output `systemctl --user`. Một agent kiểm tra cái sai sẽ báo “service not found” hoặc “service inactive” khi service thực sự đang chạy tốt ở cấp system.

Phòng ngừa: luôn chỉ định cấp độ service trong mô tả tác vụ.

```
XẤU: "Kiểm tra nginx có đang chạy không và khởi động lại nếu không"
TỐT: "Kiểm tra nginx có đang chạy như một SYSTEM service (sudo systemctl status nginx)
và khởi động lại nếu không. KHÔNG dùng cờ --user."
```

Chế Độ Thất Bại 3: Thất Bại Thâm Lặng Khi Thiếu Thư Mục

Đây là cái đau nhất: một lần triển khai Firecrawl khi sub-agent được giao nhiệm vụ triển khai các file cấu hình vào `/opt/firecrawl/config/`.

Thư mục không tồn tại.

Sub-agent cố ghi các file, nhận lỗi “No such file or directory”, và — thay vì thất bại to tiếng — ghi nó như cảnh báo, đánh dấu tác vụ hoàn thành trong output, và thoát với code 0.

Orchestrator thấy “SUCCESS” và chuyển sang tác vụ tiếp theo (khởi động dịch vụ Firecrawl). Dịch vụ không khởi động được vì các file cấu hình không có mặt ở đó.

```
# Debug sự cố sau thực tế
# Output của sub-agent về mặt kỹ thuật là đúng – nó báo cáo những gì đã xảy ra
# Nhưng "những gì đã xảy ra" bị chôn vùi trong một dòng văn bản

grep -n "No such file\|WARN\|ERROR\|failed" /tmp/subagent-firecrawl-deploy.log
# Dòng 847: WARN: Không thể ghi /opt/firecrawl/config/app.json: No such file or directory
# Dòng 848: Bỏ qua triển khai file cấu hình, service có thể không khởi động được
# Dòng 1203: Tác vụ hoàn thành. 3/4 bước thành công.
```

Cách sửa là xác thực. Trước khi sub-agent khai báo thành công, nó phải xác minh công việc của chính mình.

```
# sub-agent-validator.py
# Các mẫu xác thực mà sub-agent nên sử dụng trước khi báo cáo hoàn thành

import os
from pathlib import Path

class TaskValidator:
    """Các kiểm tra xác thực mà sub-agent nên chạy trước khi báo cáo hoàn thành."""

    @staticmethod
    def verify_files_written(expected_files: list[str]) -> tuple[bool, list[str]]:
```

```

"""Xác minh các file đã thực sự được ghi."""
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} (rỗng)")
return len(missing) == 0, missing

@staticmethod
def verify_directory_exists(dirpath: str) -> bool:
    """Xác minh thư mục tồn tại trước khi cố ghi vào."""
    return Path(dirpath).is_dir()

@staticmethod
def verify_service_running(service_name: str, user_level: bool = False) -> bool:
    """Xác minh một systemd service đang thực sự chạy."""
    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]]:
    """Chạy kiểm tra trước khi bắt đầu tác vụ."""
    errors = []

    # Kiểm tra các thư mục đích tồn tại trước khi ghi
    for target_dir in task.get("target_directories", []):
        if not Path(target_dir).exists():
            errors.append(f"Thư mục đích không tồn tại: {target_dir}")
            # Thử tạo nó
            try:
                Path(target_dir).mkdir(parents=True, exist_ok=True)
                errors[-1] += " (đã tạo)"
                errors.pop() # Xóa lỗi nếu tạo thành công
            except PermissionError:
                errors[-1] += " (bị từ chối quyền, không thể tạo)"

    # Kiểm tra các file bắt buộc tồn tại
    for required_file in task.get("required_files", []):
        if not Path(required_file).exists():
            errors.append(f"File bắt buộc thiếu: {required_file}")

    return len(errors) == 0, errors

```

Quy tắc chúng tôi hiện thực thi: Mô tả tác vụ sub-agent phải bao gồm một bước xác minh rõ ràng.

TÁC VỤ: Triển khai file cấu hình Firecrawl

Các bước:

1. Đảm bảo thư mục /opt/firecrawl/config/ tồn tại (tạo nếu thiếu)
2. Ghi app.json, redis.json, và env.json vào thư mục đó
3. XÁC MINH: xác nhận cả ba file tồn tại và không rỗng

4. Chỉ báo cáo SUCCESS nếu tất cả ba file được xác minh có mặt

Nếu tạo thư mục thất bại do quyền, báo cáo FAILED với chi tiết.
KHÔNG báo cáo thành công nếu bất kỳ file nào bị thiếu.

Chuỗi Fallback Provider

Thiết Lập Ba Provider

Chúng tôi chạy ba cấu hình LLM provider theo thứ tự ưu tiên:

API Proxy Service → Anthropic Native → ZAI

- **API Proxy Service:** Proxy cục bộ của chúng tôi định tuyến đến Anthropic, thêm request logging, thực thi giới hạn tốc độ, và xử lý tổng hợp hóa đơn
- **Anthropic Native:** API Anthropic trực tiếp — fallback nếu proxy down
- **ZAI:** Dịch vụ API bên thứ ba — fallback nếu Anthropic native gặp sự cố

```
# provider-fallback-chain.py
# LLM provider với fallback tự động khi thất bại

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 tên model chuẩn sang tên dành riêng cho provider
    priority: int # Thấp hơn = ưu tiên cao hơn

PROVIDERS = [
    ProviderConfig(
        name="api-proxy",
        base_url=os.environ.get("API_PROXY_URL", "http://10.8.0.5:4000"),
        api_key_env="API_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 dùng tên khác
        "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} đánh dấu thất bại trong {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 quá tải - cooldown ngắn
                    self._mark_failed(provider, cooldown=120)
                elif "401" in error_str or "403" in error_str:
                    # Lỗi xác thực - cooldown dài hơn, kiểm tra credentials
                    self._mark_failed(provider, cooldown=3600)
                print(f"Lỗi xác thực cho {provider.name}: kiểm tra {provider.api_key_env}")

```

```

else:
    # Lỗi không xác định – cooldown ngắn, thử cái tiếp theo
    self._mark_failed(provider, cooldown=60)
    print(f"Lỗi với {provider.name}: {e}")

return None # Tất cả provider thất bại

def _call_provider(self, provider, model, messages, api_key, **kwargs) -> str:
    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

```

Giao Tiếp Agent Đa Server

Mesh Dựa Trên VPN

Các agent của chúng tôi không phải tất cả đều chạy trên cùng một server. Gateway agent chạy trên workstation chính. Một số batch job chạy trên các compute server chuyên dụng. Monitoring agent chạy trên Server-3.

Giao tiếp diễn ra qua VPN mesh WireGuard. Tất cả server đều nằm trên **10.8.0.0/24**. Các agent tìm thấy nhau qua VPN IP.

```

# /etc/wireguard/wg0.conf (đơn giản hóa)
[Interface]
PrivateKey = <đã ẩn>
Address = 10.8.0.1/24
ListenPort = 51820

# Workstation (gateway agent)
[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

```

Để ủy thác tác vụ đa server, các agent dùng HTTP API nhẹ thay vì hệ thống file chung (hệ thống file không được chia sẻ qua các server vật lý):

```
# cross-server-agent-api.py
# HTTP API đơn giản để ủy thác tác vụ đa server

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:
    """Xác minh chữ ký HMAC trên các request đến."""
    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(...),
):
    # Xác minh request đến từ một agent đáng tin
    payload = f"{request.task}:{request.model}"
    if not verify_signature(payload, x_signature):
        raise HTTPException(status_code=401, detail="Chữ ký không hợp lệ")

    task_id = f"task-{int(asyncio.get_event_loop().time())}"

    # Tạo sub-agent bất đồng bộ
    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):
    """Chạy một tác vụ sub-agent và lưu kết quả."""
    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
    )

    # Lưu kết quả để truy xuất
    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)

```

Phục Hồi Mesh Agent: Điều Phối Phân Tán Dựa Trên Flock

Vấn Đề: Hai Agent, Cùng Tác Vụ

Trong một thiết lập đa agent, hai agent có thể độc lập quyết định làm việc trên cùng một tác vụ. Điều này ổn với các tác vụ chỉ đọc. Với các thao tác ghi, đây là thảm họa — hai agent đồng thời ghi các file cấu hình, khởi động lại service, hoặc chạy migration.

Chúng tôi dùng file locking qua `flock` để điều phối quyền truy cập độc quyền vào các tài nguyên.

```

#!/bin/bash
# agent-lock.sh
# Lấy khóa độc quyền trước khi thực hiện các thao tác ghi
# Cách dùng: agent-lock.sh <tên-khóa> <lệnh>

LOCK_NAME="${1:-default}"
LOCK_FILE="/tmp/agent-locks/${LOCK_NAME}.lock"

```

```

LOCK_TIMEOUT="${AGENT_LOCK_TIMEOUT:-300}" # Mặc định 5 phút

mkdir -p /tmp/agent-locks

# Thử lấy khóa với timeout
if flock --exclusive --timeout "$LOCK_TIMEOUT" 9; then
    echo "Đã lấy khóa: $LOCK_NAME"
    # Chạy lệnh
    shift # Xóa tên-khóa khỏi args
    "$@"
    EXIT_CODE=$?
    echo "Lệnh hoàn thành, giải phóng khóa: $LOCK_NAME"
    exit $EXIT_CODE
else
    echo "LỖI: Không thể lấy khóa '$LOCK_NAME' trong ${LOCK_TIMEOUT}s"
    echo "Một agent khác có thể đang làm việc trên tài nguyên này"
    exit 1
fi 9>"$LOCK_FILE"

# distributed-lock.py
# Context manager Python cho việc khóa agent phân tán

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 cho việc truy cập độc quyền của agent vào một tài nguyên.

    Cách dùng:
    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"Đã khóa bởi PID {os.getpid()} lúc {time.time()}\n")
                lock_fd.flush()
                break
            except IOError:
                elapsed = time.time() - start
                print(f"Đang chờ khóa '{resource_name}' ({elapsed:.0f}s/{timeout}s)...")
                time.sleep(5)

    if not acquired:

```

```

        raise TimeoutError(
            f"Không thể lấy khóa '{resource_name}' trong {timeout}s. "
            f"Một agent khác có thể đang giữ nó."
        )

    print(f"Đã lấy khóa: {resource_name}")
    yield

finally:
    if acquired:
        fcntl.flock(lock_fd, fcntl.LOCK_UN)
        print(f"Đã giải phóng khóa: {resource_name}")
    lock_fd.close()

# Sử dụng trong tác vụ sub-agent
async def deploy_configuration(config_files: dict):
    with agent_lock("firecrawl-config-deployment", timeout=120):
        # Chỉ một agent có thể triển khai cấu hình Firecrawl tại một thời điểm
        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"Đã ghi: {target}")

```

Tự Phục Hồi: Phát Hiện và Khôi Phục Từ Các Agent Bị Treo

```

# agent-health-monitor.py
# Theo dõi sức khỏe sub-agent và khôi phục các tác vụ bị treo

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 # Tác vụ chạy lâu hơn thế này là đáng ngờ

def log_task_start(task_id: str, task_description: str, pid: int):
    """Ghi log bắt đầu tác vụ để theo dõi."""
    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):
    """Ghi log hoàn thành tác vụ."""
    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]:
    """Tìm các tác vụ đã chạy lâu hơn ngưỡng."""
    if not TASK_LOG.exists():
        return []

    # Xây dựng trạng thái tác vụ từ 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

    # Kiểm tra tác vụ đang chạy nào bị treo
    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:
            # Kiểm tra tiến trình có còn chạy không
            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_time).seconds // 60)
                except (psutil.NoSuchProcess, psutil.AccessDenied):
                    # Tiến trình đã biến mất nhưng tác vụ chưa được ghi là kết thúc
                    log_task_end(task_id, "abandoned")

    return stalled

def recover_stalled_task(task: dict):
    """Thử khôi phục một tác vụ bị treo."""
    pid = task.get("pid")
    task_id = task["task_id"]

    print(f"Đang khôi phục tác vụ bị treo {task_id} (PID {pid}, đã chạy {task['running_minutes']}
phút)")

    # Kill tiến trình bị treo
    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"Không thể kill PID {pid}: {e}")

    # Giải phóng các khóa mà tác vụ này đang giữ

```

```

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"Đã giải phóng khóa: {lock_file.name}")
    except Exception:
        pass

log_task_end(task_id, "recovered")
print(f"Tác vụ {task_id} đã khôi phục. Khuyến nghị đưa vào hàng đợi lại: {task['description']}")

```

Thực Hành Tốt Nhất Cho Sub-Agent: Runbook Vận Hành

Sau một năm chạy infrastructure đa agent, đây là runbook vận hành chúng tôi chia sẻ cho những người mới tham gia hệ thống.

Nên Làm

- [✓] Luôn bao gồm --cleanup delete khi tạo sub-agent
- [✓] Chỉ định model rõ ràng – không bao giờ dựa vào mặc định cho tác vụ infra
- [✓] Bao gồm bước xác minh trong mọi tác vụ ghi
- [✓] Dùng flock hoặc tương đương trước bất kỳ thao tác ghi nào
- [✓] Đặt timeout rõ ràng – không bao giờ để sub-agent chạy vô thời hạn
- [✓] Ghi log bắt đầu và kết thúc tác vụ với task ID để theo dõi sức khỏe
- [✓] Bao gồm thư mục làm việc rõ ràng trong mô tả tác vụ
- [✓] Với system service, chỉ định cấp system vs user trong tác vụ
- [✓] Kiểm tra tác vụ sub-agent trên staging trước khi chạy production
- [✓] Phạm vi tác vụ phù hợp với ~100K token context (để lại 50K headroom)

Không Nên Làm

- [x] Đừng giả định "success" từ sub-agent mà không xác minh đầu ra
- [x] Đừng tạo sub-agent cho các tác vụ có thể hoàn thành trong <10s
- [x] Đừng dùng Gemini cho các thay đổi infrastructure
- [x] Đừng để sub-agent tích lũy – kiểm tra danh sách phiên thường xuyên
- [x] Đừng chia sẻ dataset giữa các agent mà không có cô lập rõ ràng
- [x] Đừng hardcode địa chỉ server – dùng VPN hostname hoặc env var
- [x] Đừng chạy migration hoặc thay đổi schema mà không có khóa flock
- [x] Đừng tin vào output sub-agent không bao gồm xác minh rõ ràng
- [x] Đừng tạo sub-agent trong các vòng lặp mà không có giới hạn tốc độ

Template Mô Tả Tác Vụ

```

## Tác vụ: [Tiêu đề ngắn gọn]

**Model:** claude-opus-4-6 [hoặc chỉ định]
**Thư mục làm việc:** /đường/dẫn/đến/workdir
**Cấp độ service:** system [hoặc user, cho tác vụ systemd]
**Timeout:** 20 phút

```

```
### Kiểm Tra Trước Khi Bắt Đầu
Trước khi bắt đầu, xác minh:
- [ ] [Thư mục X] tồn tại (tạo nếu thiếu với quyền thích hợp)
- [ ] [Service Y] hiện đang [chạy/dừng]
- [ ] [File Z] có thể đọc được

### Các Bước
1. [Bước cụ thể, không mơ hồ]
2. [Bước cụ thể, không mơ hồ]
3. [Bước cụ thể, không mơ hồ]

### Xác Minh
Sau khi hoàn thành tất cả các bước, xác nhận:
- [ ] [File mong đợi] tồn tại và không rỗng
- [ ] [Service mong đợi] đang hoạt động (dùng: sudo systemctl is-active [service])
- [ ] [Endpoint mong đợi] phản hồi (dùng: curl -f http://...)

### Báo Cáo
Ghi vào ~/.agent-gateway/workspace/inbox/from-claudecode.md:
- Trạng thái: SUCCESS hoặc FAILED
- Những gì đã làm
- Bất kỳ vấn đề gặp phải
- Các file đã sửa đổi (với đường dẫn)
```

Bài Học Rút Ra

“Success” từ sub-agent có nghĩa là agent đã kết thúc, không phải tác vụ đã thành công. Tích hợp xác minh rõ ràng vào mọi tác vụ. Coi báo cáo của agent là gợi ý, không phải sự thật.

Thất bại giới hạn context là thâm lặng. Một sub-agent chạm giới hạn context không crash — nó suy giảm. Nó tạo ra các phản hồi ngắn hơn, bỏ qua các bước, và có thể báo cáo thành công trên một tác vụ chưa hoàn thành. Phạm vi tác vụ để sử dụng ít hơn hai phần ba context window.

Lựa chọn model là quyết định chính sách, không phải sở thích. Thực thi nó theo chương trình. Người hiểu tại sao Gemini không thể tin tưởng cho quản lý service cấp hệ thống có thể không trực ban khi cảnh báo kích hoạt.

Dọn dẹp không phải tùy chọn. -cleanup delete, giải phóng flock, dọn dẹp file khóa — tất cả mọi thứ. Các phiên mô cô và khóa cũ tích lũy theo thời gian thành các sự cố vận hành.

Shared workspace có công nghệ thấp và đáng tin cậy. Chúng tôi đã đánh giá một số message broker cho giao tiếp agent. Chúng tôi đã triển khai phương pháp thư mục chung. Các file phẳng sống sót qua việc khởi động lại server, không có vấn đề về tính khả dụng của broker, và dễ dàng kiểm tra bởi bất kỳ công cụ nào.

Giao tiếp đa server cần xác thực. Các agent chạy trên các server khác nhau giao tiếp qua VPN vẫn cần ký request. VPN cung cấp cô lập cấp mạng; ký HMAC cung cấp danh tính cấp ứng dụng. Cả hai đều cần thiết.

Điều phối đa agent không phải là bài toán đã được giải quyết. Mỗi hệ thống có các chế độ thất bại riêng, ranh giới tác vụ tối ưu riêng, chính sách model riêng. Những gì hoạt động với chúng tôi — tmux + shared workspace + flock + xác minh rõ ràng — hoạt động vì nó phù hợp với infrastructure cụ thể của chúng tôi. Thiết lập của bạn sẽ khác.

Các nguyên tắc phổ quát: thất bại to tiếng, xác minh rõ ràng, phạm vi hẹp, và không bao giờ tin vào báo cáo của sub-agent mà không kiểm tra công việc của nó.

Kết thúc Phần 4: Các Chủ Đề Nâng Cao.

Phụ lục A đề cập đến các kiến trúc tham khảo. Phụ lục B đề cập đến bộ công cụ đầy đủ được sử dụng xuyên suốt cuốn sách này.

PHẦN 5: TƯƠNG LAI

Chương 15: Mở Rộng Quy Mô Hệ Thống Agent

“Agent đầu tiên là bằng chứng khái niệm. Agent thứ mười là cơn ác mộng vận hành. Agent thứ một trăm là lợi thế cạnh tranh hoặc đồng tiền đang bốc cháy — thường là cả hai.”

Giới Thiệu

Khi tôi triển khai AI agent đầu tiên trong môi trường thực tế, tôi nghĩ mình đã vượt qua phần khó nhất. Tôi có một agent hoạt động, một server ổn định, và sự lạc quan hồi hộp rằng mọi thứ sẽ... vận hành được. Sáu tháng sau, tôi có mười bảy agent chạy trên bốn server, một Patroni cluster đã failover hai mươi hai lần trong một cuối tuần, và hóa đơn đám mây khiến tay tôi run.

Mở rộng quy mô hệ thống agent không giống mở rộng các microservice truyền thống. Các chế độ thất bại kỳ lạ hơn. Chi phí khó dự đoán hơn. Và việc debug mang tính triết học hơn — “tại sao agent quyết định rằng khởi động lại database là phản ứng đúng cho một truy vấn chậm?” không phải câu hỏi bạn mong đợi phải đặt ra lúc 2 giờ sáng.

Chương này là chương tôi ước mình đã có trước khi bắt đầu mở rộng. Nó đề cập đến các thực tế kỹ thuật: tính khả dụng cao của database, cân bằng tải, hệ thống bộ nhớ, và quản lý chi phí. Nhưng nó cũng đề cập đến các thực tế vận hành mà các whitepaper không bao giờ đề cập — như tại sao tôi cuối cùng đã xóa mọi cron job tự động trong hệ thống, và tại sao đó là quyết định đúng đắn.

Phần 1: Từ Một Agent Đến Một Mesh Agent

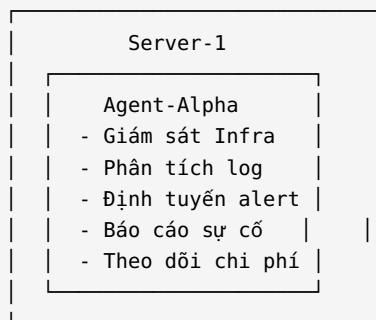
Giai Đoạn Một Agent (Ngày 1-30)

Agent đầu tiên của bạn rất đơn giản. Nó chạy trên một server, dùng một model, và có một nhiệm vụ. Bạn có thể suy nghĩ về nó trong đầu. Khi nó hỏng, bạn biết phải nhìn vào đâu.

Giai đoạn này kết thúc khi agent hoạt động đủ tốt đến mức bạn muốn nó làm nhiều hơn. Đó là cái bẫy.

Bản năng đúng đắn — “cái này hoạt động, hãy thêm khả năng hơn” — dẫn thẳng đến kiến trúc sai: một agent làm mười việc kém thay vì một việc tốt.

Kiến trúc một agent (đơn giản nhưng dễ vỡ)



Agent trở nên chậm vì nó đang làm quá nhiều. Context window đầy. Model bắt đầu đưa ra quyết định kém hơn vì prompt giờ là mười lăm nghìn token context hỗn hợp. Bạn thêm bộ nhớ. Nó chậm hơn.

Điểm uốn: Khi độ trễ phản hồi trung bình của agent vượt quá 15 giây và bạn thấy mình nghĩ “tôi sẽ chỉ thêm hướng dẫn để sửa cái này,” đã đến lúc tách ra.

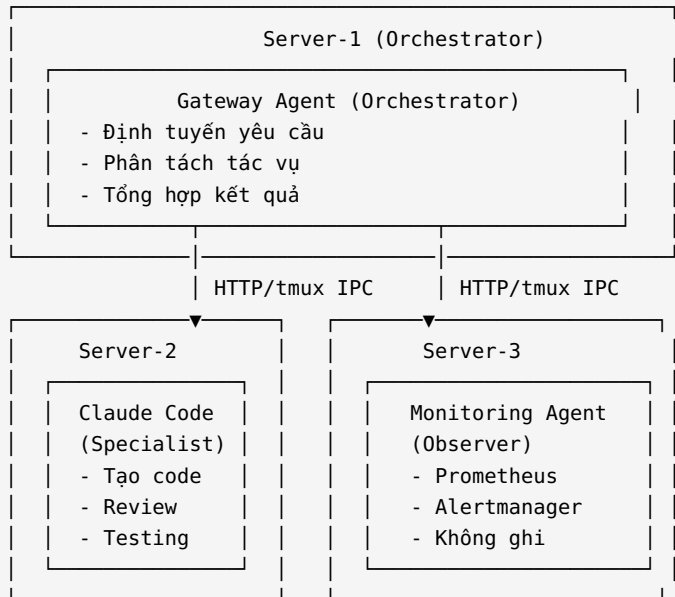
Thiết Kế Mesh Agent (Ngày 30-90)

Mesh không chỉ là nhiều agent — đó là một cấu trúc liên kết có chủ đích trong đó các agent có ranh giới sở hữu rõ ràng và các giao thức giao tiếp được xác định rõ.

Mesh khả thi tối thiểu có ba vai trò:

1. **Orchestrator** — nhận yêu cầu, phân tách tác vụ, ủy thác cho các chuyên gia, tổng hợp kết quả
2. **Specialists** — khả năng sâu trong một lĩnh vực (infra, code, data, v.v.)
3. **Observers** — giám sát thụ động, không có quyền ghi, chỉ báo cáo

Cấu trúc liên kết mesh agent (tối thiểu 3 agent)



Giao tiếp giữa các agent: Tôi đã thử ba phương pháp trước khi chọn được một cái hoạt động.

- **REST API** — sạch sẽ nhưng thêm độ trễ và yêu cầu mỗi agent chạy một HTTP server. Overhead khoảng ~200ms mỗi hop, cộng dồn rất tệ trong các chuỗi.
- **Message queue (Redis Streams)** — tốt hơn cho các tác vụ bất đồng bộ, nhưng thêm độ phức tạp hạ tầng và một thứ nữa phải giám sát.
- **IPC dựa trên tmux** — không phải những gì bạn mong đợi được khuyến nghị trong một cuốn sách, nhưng nó hoạt động. Mỗi agent chạy trong một phiên tmux được đặt tên. Orchestrator gửi lệnh bằng cách ghi vào một file chung mà agent đích poll, hoặc dùng `tmux send-keys`. Độ trễ dưới 50ms cho các cuộc gọi cục bộ. Không có hạ tầng thêm.

```
# Gửi tác vụ từ Orchestrator đến Claude Code agent qua 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
}

# Vòng lặp polling của Claude Code agent
poll_inbox() {
    local inbox_file="$HOME/.agents/inbox/claude-code.json"
    while true; do
        if [[ -f "$inbox_file" ]]; then
            task=$(cat "$inbox_file")
        fi
    done
}
```

```

    rm "$inbox_file"
    process_task "$task"
fi
sleep 2
done
}

```

Điều này không hào nhoáng. Nó cũng không dễ vỡ. Các phiên tmux sống sót qua việc ngắt kết nối SSH. Các file không cần một service đang chạy. Khi mọi thứ bốc lửa, bạn vẫn có thể kiểm tra thư mục inbox bằng `ls`.

Quản Trị Mesh: Ai Sở Hữu Cái Gì

Thất bại phổ biến nhất trong các hệ thống đa agent không phải là kỹ thuật — đó là nhầm lẫn sở hữu. Hai agent đều cố sửa cùng một alert. Không agent nào kiểm tra xem agent kia đã hành động chưa. Hệ thống cấp phát kép, gửi kép, tính phí kép.

Triển khai một hệ thống lease đơn giản trước khi bạn có nhiều hơn hai agent:

```

# Lease phân tán đơn giản dùng PostgreSQL advisory locks
import psycopg2
import hashlib

def acquire_task_lease(task_id: str, agent_id: str, ttl_seconds: int = 300) -> bool:
    """
    Trả về True nếu agent này đã lấy được lease cho task_id.
    Dùng PostgreSQL advisory locks để đảm bảo tính nguyên tử.
    """
    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:
                # Ghi lại ai đang giữ 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

```

Phần 2: Mở Rộng Ngang

Khi Nào Thêm Server

Thêm server tốn tiền. Không thêm server khi cần thì tốn nhiều tiền hơn (về độ trễ agent, SLA bị bỏ lỡ, và các buổi tối của bạn). Tiêu chí quyết định tôi dùng:

Tín hiệu	Ngưỡng	Hành động
Độ sâu hàng đợi agent	> 10 tác vụ đang chờ	Thêm server chuyên gia
Độ trễ phản hồi trung bình	> 20s kéo dài	Tách khối lượng công việc
Áp lực bộ nhớ	> 85% RAM liên tục	Thêm server hoặc giảm agent
Chi phí mỗi tác vụ	> \$0.05 trung bình	Kiểm toán và tối ưu trước

Đừng thêm server để giải quyết vấn đề chi phí. Bạn sẽ có hai server và cùng vấn đề chi phí.

Baseline Ba Server

Khi bạn có một mesh đạt chuẩn production, bạn có thể sẽ hội tụ về ít nhất ba server:

```
Server-1 (Orchestrator / Control Plane)
- Agent orchestrator
- Database proxy (PgBouncer)
- Reverse proxy (Traefik)
- Monitoring stack (Prometheus, Grafana, Alertmanager)
- RAM: tối thiểu 16GB
- CPU: tối thiểu 4 core

Server-2 (AI Workload / Hot Path)
- Các specialist agent tần số cao
- Storage nhanh (NVMe)
- RAM: 32GB+ (agent ngôn bộ nhớ)
- CPU: 8+ core
- Lưu ý: GPU tùy chọn nhưng hiếm khi tiết kiệm chi phí cho các agent dựa trên API

Server-3 (Database / Persistence Layer)
- PostgreSQL primary (quản lý bởi Patroni)
- Cognition knowledge graph
- Backup agent
- RAM: tối thiểu 16GB (nhiều hơn nếu Cognition được dùng nhiều)
- Storage: IOPS cao, volume riêng cho WAL
```

Server-3 của tôi là một instance ARM từ nhà cung cấp đám mây đã chạy khoảng tám năm mà không có sự cố ngoài kế hoạch. Điều này thực sự ấn tượng. Nó cũng chậm đối với AI workload — khoảng 26 giây mỗi yêu cầu

inference khi tôi thử chạy model cục bộ trên đó. Kiến trúc ARM chạy các thao tác database hiệu quả nhưng gặp khó khăn với phép tính floating-point nặng mà LLM inference đòi hỏi. Bài học: ARM xuất sắc cho database, không phải cho AI compute. Đừng nhầm lẫn lịch sử uptime với tính phù hợp đa năng.

Cân Bằng Tải Các Yêu Cầu Agent

Đối với các endpoint agent dựa trên HTTP, Traefik là công cụ phù hợp. Nó xử lý service discovery, TLS termination, và weighted routing mà không cần bạn viết cấu hình routing bằng tay.

```
# 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 # Server có năng lực hơn, trọng số cao hơn

  middlewares:
    agent-ratelimit:
      rateLimit:
        average: 100
        burst: 50
        period: "1m"
        sourceCriterion:
          ipStrategy:
            depth: 1

    agent-retry:
      retry:
        attempts: 3
        initialInterval: "500ms"
```

Đối với các agent dựa trên hàng đợi tác vụ (phương pháp tmux IPC), cân bằng tải đơn giản hơn — orchestrator duy trì một pool các worker khả dụng và gán tác vụ round-robin với lọc khả năng:

```

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:
        """Trả về agent_id của agent được gán, hoặc None nếu không có agent khả dụng."""
        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

            # Gán cho agent ít tải nhất
            chosen = min(candidates, key=lambda a: a.active_tasks)
            chosen.active_tasks += 1
            return chosen.id

```

Phần 3: Tính Khả Dụng Cao Của Database — Các Bài Học Khó

Phần này là phần tôi mong muốn nhất đã đọc trước khi xây dựng. Database HA cho các hệ thống agent không khác biệt đáng kể so với HA cho bất kỳ ứng dụng nào khác — nhưng các agent tương tác với database theo những cách khiến lộ ra các chế độ thất bại mà bạn sẽ không gặp với người dùng con người.

Patroni + etcd: Kiến Trúc

Patroni quản lý PostgreSQL failover. etcd cung cấp sự đồng thuận phân tán mà Patroni dùng để bầu chọn primary. Bạn cần ít nhất ba node etcd cho quorum đúng đắn — với hai node, bất kỳ phân vùng mạng nào cũng khiến cả hai node từ chối làm primary, và cluster của bạn trở thành chỉ đọc.

```

# Đúng: 3 etcd node = chịu được 1 node thất bại
etcd-1 (Server-1) ————|
etcd-2 (Server-2) ————|—— Patroni cluster

```

etcd-3 (Server-3) ———┘

Sai: 2 etcd node = bất kỳ phân vùng nào = split-brain
etcd-1 — etcd-2 (đừng làm vậy)

Cấu hình etcd cho mỗi node:

```
# /etc/etcd/etcd.conf.yml - ví dụ node 1
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

# Điều chỉnh heartbeat và election
heartbeat-interval: 100
election-timeout: 1000

# Compaction - quan trọng cho cluster chạy lâu dài
auto-compaction-mode: periodic
auto-compaction-retention: "1h"
```

Cuối Tuần 22 Failover: Một Postmortem

Một sáng thứ Bảy, monitoring của tôi bắt đầu kích hoạt. Patroni đang failover. Rồi lại. Rồi lại. Đến tối Chủ nhật, tôi có 22 sự kiện failover được ghi trong lịch sử của Patroni. Database về mặt kỹ thuật vẫn khả dụng — nó tiếp tục bầu chọn primary mới — nhưng mỗi failover gây ra sự ngừng hoạt động 10-30 giây cho các agent. Các background job đang thất bại. Các agent đang thử lại. Các lần thử lại đang tạo ra token. Token đang tổn tiền.

Nguyên nhân gốc: Lỗi cấu hình etcd.

Khi tôi đã thêm Server-3 vào cluster ba tuần trước, tôi đã cập nhật chuỗi `initial-cluster` trên server 1 và 3 nhưng không cập nhật nó trên server 2. Tài liệu etcd lưu ý rằng `initial-cluster` chỉ được dùng trong quá trình bootstrapping, điều đó về mặt kỹ thuật là đúng — nhưng nó ảnh hưởng đến cách các node đối chiếu danh sách peer của chúng sau khi khởi động lại.

Server-2 đã khởi động lại trong quá trình bảo trì thông thường, tham gia lại với danh sách peer cũ, và tạo ra sự không nhất quán dai dẳng trong cách ba node nhìn thấy nhau.

Trong tải bình thường, điều này vô hình. Dưới tải cao hơn từ một lần triển khai ngày thứ Bảy, nó bề mặt như các cuộc bầu chọn leader giả.

Cách sửa — từng bước:

```
# Bước 1: Xác định node không khớp
for server in server1 server2 server3; do
    echo "=== $server ==="
    ssh $server "etcdctl --endpoints=http://127.0.0.1:2379 member list"
done

# Bước 2: Dừng Patroni trên tất cả node TRƯỚC (quan trọng: dừng Patroni trước etcd)
for server in server1 server2 server3; do
    ssh $server "sudo systemctl stop patroni"
done

# Bước 3: Dừng etcd trên node không khớp
ssh server2 "sudo systemctl stop etcd"

# Bước 4: Xóa dữ liệu etcd trên node không khớp (không phá hủy dữ liệu Postgres)
ssh server2 "sudo rm -rf /var/lib/etcd/data /var/lib/etcd/wal"

# Bước 5: Đồng bộ cấu hình etcd – mỗi node phải có initial-cluster giống hệt nhau
for server in server1 server2 server3; do
    ssh $server "sudo diff /etc/etcd/etcd.conf.yml /etc/etcd/etcd.conf.yml.reference"
done

# Bước 6: Khởi động lại etcd theo thứ tự – thành viên hiện tại trước, thành viên mới sau
ssh server1 "sudo systemctl start etcd"
ssh server3 "sudo systemctl start etcd"
sleep 10 # Để quorum thiết lập
ssh server2 "sudo systemctl start etcd"

# Bước 7: Kiểm tra sức khỏe cluster trước khi khởi động Patroni
etcdctl --endpoints=http://server1:2379,http://server2:2379,http://server3:2379 \
    endpoint health

# Bước 8: Khởi động lại Patroni
for server in server1 server2 server3; do
    ssh $server "sudo systemctl start patroni"
    sleep 5 # Phân tán để tránh thundering herd
done

# Bước 9: Xác minh Patroni đã bầu chọn một primary duy nhất
patronictl -c /etc/patroni/patroni.yml list
```

Phòng ngừa: Thêm kiểm tra đồng bộ cấu hình vào pipeline triển khai. Mỗi node etcd nên có danh sách peer giống hệt nhau. Nếu chúng khác nhau, việc triển khai thất bại.

```
#!/bin/bash
# etcd-config-check.sh – chạy trước mỗi lần triển khai
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 "THẤT BẠI: Lỗi cấu hình etcd trên $node"
        echo "  Mong đợi: $REFERENCE_HASH"
        echo "  Nhận được: $NODE_HASH"
        exit 1
    fi
done

echo "THÀNH CÔNG: Cấu hình etcd nhất quán trên tất cả node"
```

Cấu Hình Patroni (Hardened cho Production)

```
# /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 – không failover nếu replica tụt hậu quá xa
  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'

```

Chiến lược Backup: Ba Kho Lưu Trữ

Đối với database hệ thống agent, một đích backup duy nhất là không đủ. Các agent đọc và ghi liên tục. Một backup bị hỏng mà bạn phát hiện trong quá trình khôi phục là trường hợp tệ nhất.

Chiến lược backup production của tôi dùng ba đích độc lập:

Backup Target 1: Nhà cung cấp storage tương thích S3 (chính, datacenter cục bộ)

- Full backup: hàng ngày lúc 02:00
- WAL archiving: liên tục
- Retention: 30 ngày
- Chi phí ước tính: ~\$8/tháng cho 30GB

Backup Target 2: Object Storage nhà cung cấp đám mây (phụ, đám mây khác)

- Full backup: hàng ngày lúc 03:00
- WAL archiving: liên tục
- Retention: 14 ngày
- Chi phí ước tính: \$0 (free tier của nhà cung cấp đám mây)

Backup Target 3: MinIO (tự host, on-premises)

- Full backup: hàng tuần
- Retention: 90 ngày
- Chi phí ước tính: điện + disk (~\$2/tháng)

Các con số về nén đáng hiểu. Full backup của database production (hội thoại agent, metadata knowledge graph, lịch sử tác vụ, bản ghi chi phí) nén như sau:

Thư mục dữ liệu PostgreSQL thô:	561.7 MB
pg_basebackup (tar):	418.3 MB
pgBackRest (nén lz4):	30.2 MB
Tỷ lệ nén:	18.6:1

Nén LZ4 trên dữ liệu PostgreSQL hiệu quả đáng kể vì database hệ thống agent chứa nhiều cấu trúc JSON lặp lại và tên model, tên công cụ, và thông báo lỗi lặp lại. Luôn bật nén trên công cụ backup của bạn.

```

# /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=primary-db-backups
repo2-s3-endpoint=${S3_PROVIDER_ENDPOINT}
repo2-s3-region=${S3_PROVIDER_REGION}
repo2-s3-key=${S3_PRIMARY_KEY}
repo2-s3-key-secret=${S3_PRIMARY_SECRET}

repo3-type=s3
repo3-path=/pgbackrest-prod
repo3-s3-bucket=secondary-db-backups
repo3-s3-endpoint=${CLOUD_PROVIDER_S3_ENDPOINT}
repo3-s3-region=${CLOUD_PROVIDER_REGION}
repo3-s3-key=${CLOUD_S3_KEY}
repo3-s3-key-secret=${CLOUD_S3_SECRET}

# Lịch backup (trong 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

```

Phần 4: Mở Rộng Bộ Nhớ và Tri Thức

Vấn Đề Với Bộ Nhớ Agent Chung

Các agent cần bộ nhớ. Không có bộ nhớ, mỗi cuộc hội thoại bắt đầu từ số không — agent không thể nhớ rằng bạn đã nói với nó rằng infrastructure của bạn dùng Kubernetes, hoặc staging environment có vấn đề đã biết với auth service. Điều này không chỉ bất tiện; nó có nghĩa là các agent lặp lại công việc khởi tạo tốn kém trong mỗi lần gọi.

Giải pháp ngay thơ là một database chung. Mỗi agent đọc và ghi vào cùng một bộ nhớ. Điều này hoạt động cho đến khi không hoạt động — khoảng thời gian bạn có ba agent đồng thời ghi tri thức mâu thuẫn về cùng một thành phần infrastructure.

Cognee Trên Nhiều Server

Cognee là một hệ thống knowledge graph được thiết kế cho AI agent. Nó cung cấp bộ nhớ có cấu trúc với các quan hệ đồ thị, tìm kiếm ngữ nghĩa, và phiên bản hóa thời gian. Mở rộng nó trên nhiều server đòi hỏi chú ý cẩn thận đến mô hình nhất quán đồ thị.

```
# Cấu hình Cognee đa server
import cognee
from cognee.infrastructure.databases.graph import get_graph_config

# Mỗi server kết nối đến cùng một graph backend
# nhưng dùng vector cache cục bộ theo server để tăng hiệu suất
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",
        # LanceDB cục bộ theo server để tìm kiếm vector nhanh
        "vector_db_url": f"/var/lib/cognee/{server_id}/vectors",
    })

# Cô lập namespace: các agent trên các server khác nhau
# không làm ô nhiễm knowledge graph của nhau
await cognee.config.set_llm_config({
    "namespace": f"agent-mesh-{server_id}",
})
```

Cảnh báo quan trọng: không có multi-tenancy trong Cognee (tính đến tháng 3 năm 2026). Nếu bạn có các agent cho các khách hàng hoặc dự án khác nhau chia sẻ một instance Cognee, tri thức của họ sẽ rò rỉ lẫn nhau. Agent A học rằng “production DB ở 10.0.1.1” sẽ chia sẻ tri thức đó với Agent B đang làm việc trên infrastructure của khách hàng khác. Chạy các instance Cognee riêng biệt cho mỗi tenant hoặc dự án, không phải mỗi agent.

```
# Đúng: Các instance Cognee cô lập theo dự án
├─ 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)

# Sai: Instance Cognee chung cho tất cả dự án
└─ cognee-shared/ (tri thức rò rỉ giữa alpha, beta, gamma)
```

Các Tầng Bộ Nhớ

Không phải tất cả bộ nhớ agent đều cần cùng độ bền hoặc tốc độ truy cập.
Triển khai các tầng:

Tầng 1 – Tạm thời (trong tiến trình, mất khi khởi động lại)
Dùng cho: context hội thoại hiện tại, các bước suy luận trung gian
Storage: Python dict / LRU cache
Thời gian truy cập: <1ms
Chi phí: Chỉ RAM

Tầng 2 – Phiên (Redis, dựa trên TTL)
Dùng cho: trạng thái hội thoại đa lượt, trạng thái tác vụ, sở thích người dùng
Storage: Redis với TTL (1-24 giờ)
Thời gian truy cập: 1-5ms
Chi phí: Instance Redis (~\$15/tháng được quản lý)

Tầng 3 – Bền vững (PostgreSQL)
Dùng cho: quyết định agent, lịch sử tác vụ, audit trail, bản ghi thanh toán
Storage: PostgreSQL với Patroni
Thời gian truy cập: 5-50ms
Chi phí: Bao gồm trong hạ tầng database

Tầng 4 – Knowledge Graph (Cognee/Neo4j)
Dùng cho: quan hệ thực thể, tri thức miền, các mẫu đã học
Storage: Neo4j + LanceDB (vector index)
Thời gian truy cập: 20-200ms (tìm kiếm ngữ nghĩa tốn kém)
Chi phí: \$30-100/tháng tùy kích thước đồ thị

Phần 5: Mở Rộng Chi Phí

Vấn Đề Lũy Thừa

Đây là phép tính mà không ai viết trong các tutorial “bắt đầu với AI agent”:

Một agent, 100 truy vấn/ngày:
Token trung bình mỗi truy vấn: 2.000 input + 500 output
Token hàng ngày: 250.000
Chi phí hàng ngày (Sonnet 4.6): $250K \times \$0.003/1K + 50K \times \$0.015/1K = \$1.50/\text{ngày}$

Năm agent, mỗi agent 100 truy vấn/ngày:
Nhưng các agent cũng nói chuyện với nhau: thêm 30% lưu lượng liên agent
Token hàng ngày: $5 \times 250.000 \times 1.3 = 1.625.000$
Chi phí hàng ngày: ~\$9.75/ngày

Mười agent, mỗi agent 100 truy vấn/ngày:
Lưu lượng liên agent giờ là 60% tổng (mesh bận hơn)
Token hàng ngày: $10 \times 250.000 \times 1.6 = 4.000.000$
Chi phí hàng ngày: ~\$24/ngày

Hai mươi agent, mỗi agent 100 truy vấn/ngày:
Lưu lượng liên agent 100%+ của lưu lượng trực tiếp

Token hàng ngày: $20 \times 250.000 \times 2.1 = 10.500.000$
Chi phí hàng ngày: ~\$63/ngày

Chi phí không tỷ lệ tuyến tính với số lượng agent. Nó tỷ lệ siêu tuyến tính vì các agent giao tiếp với nhau, và mỗi tin nhắn liên agent là một API call mới. Một mesh N agent có $O(N^2)$ đường giao tiếp tiềm năng. Ngay cả khi chỉ một phần nhỏ những đường đó hoạt động, hồ sơ chi phí rõ ràng là phi tuyến.

Định Cỡ Model Đúng

Đòn bẩy chi phí có tác động lớn nhất là lựa chọn model. Không phải mọi tác vụ đều cần Opus.

```
# Định tuyến model theo độ phức tạp tác vụ
def select_model(task: Task) -> str:
    # Phân loại, định tuyến, định dạng đơn giản
    if task.type in ("classify", "route", "format", "extract"):
        return "claude-haiku-4-5" # $1/$5 mỗi MTok

    # Suy luận tiêu chuẩn, tạo code, phân tích
    elif task.type in ("analyze", "generate", "review", "plan"):
        return "claude-sonnet-4-6" # $3/$15 mỗi MTok

    # Suy luận đa bước phức tạp, vấn đề mới
    elif task.complexity_score > 0.85:
        return "claude-opus-4-6" # $5/$25 mỗi MTok

    else:
        return "claude-sonnet-4-6"

# Heuristic tính điểm độ phức tạp
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)
```

Trong thực tế, sau khi định cỡ đúng, khoảng 60% truy vấn agent của tôi định tuyến đến Haiku, 35% đến Sonnet, và 5% đến Opus. Điều này giảm chi phí khoảng 70% so với chạy mọi thứ trên Sonnet.

Quyết Định Về Cron Job

Trong các giai đoạn đầu xây dựng agent mesh, tôi có 23 cron job tự động: - Tôm tắt sức khỏe infrastructure hàng giờ - Báo cáo chi phí hàng ngày - Đồng bộ knowledge graph mỗi sáu giờ - Kiểm tra heartbeat mỗi agent (mỗi 15 phút) - Phân tích log hàng đêm - Báo cáo xu hướng hàng tuần - Giám sát alert liên tục với tóm tắt được tạo bởi AI

Tôi đã xóa tất cả.

Không phải dần dần. Tất cả, trong một cuối tuần, sau một đánh giá trung thực về giá trị chúng tạo ra so với chi phí.

Phép tính rất rõ ràng. 23 cron job tập thể tiêu thụ khoảng 2,1 triệu token mỗi ngày — khoảng \$12.60/ngày, hay \$378/tháng. Giá trị chúng tạo ra là: một số báo cáo mà tôi thường không đọc, tóm tắt sức khỏe sao chép những gì Grafana đã hiển thị cho tôi, và tóm tắt alert kém có thể hành động hơn các thông báo Alertmanager thô.

Các cron job còn sống sót (tôi cuối cùng đã khôi phục bốn) là những cái trong đó: 1. AI tạo ra thứ gì đó tôi không thể lấy từ công cụ giám sát thông thường 2. Output được đọc và xử lý thường xuyên 3. Chi phí mỗi insight có thể hành động dưới \$1

Mọi thứ khác là trả tiền để tạo văn bản tích lũy trong các kênh Slack và hộp thư đến email mà không được đọc. Nếu bạn đang xây dựng agent mesh, kiểm toán các scheduled job của bạn một cách tàn nhẫn và thường xuyên. Đốt token mà không có giá trị người dùng không phải tự động hóa — đó là spam log rất đắt tiền.

Phần 6: Quản Lý Bộ Nhớ Ở Cấp Độ OS

Vấn Đề Swap

Các server phát triển agent chạy khối lượng công việc nặng: AI API client, web server, monitoring stack, và thường là công cụ build cho code mà các agent viết. Trên Server-2 của tôi, sau bất kỳ build đáng kể nào (các dự án Gradle là tệ nhất), áp lực bộ nhớ đủ nghiêm trọng để gây ra OOM kill các tiến trình agent.

Gradle và Kotlin daemon đặc biệt hung hãn. Một Gradle build điển hình để lại các tiến trình daemon tiêu thụ 3.2GB RAM — và các daemon được thiết kế để sống sót cho các build tiếp theo nhanh hơn. Trên một server phát triển cũng chạy agent, các daemon này cạnh tranh trực tiếp với các tiến trình agent đang hoạt động.

```
# Hiển thị mức sử dụng bộ nhớ Gradle daemon
./gradlew --status
# Output ví dụ:
# PID STATUS INFO
# 8421 IDLE 7.6.4
```

```
# Memory: heap=512MB, nonHeap=284MB

# Kill tất cả Gradle daemon sau khi build hoàn thành
./gradlew --stop

# Hoặc: thêm vào .gradle/gradle.properties để giới hạn thời gian sống daemon
# (trên server, không phải trong cấu hình dự án)
# org.gradle.daemon.idletimeout=3600000 # 1 giờ
# org.gradle.jvmargs=-Xmx2g -XX:MaxMetaspaceSize=512m
```

Đối với swap, tôi dùng zram — swap RAM được nén — thay vì disk swap. Disk swap trên một server agent bận rộn gây ra các đột biến độ trễ thảm khốc. zram nén dữ liệu trong RAM dùng thuật toán hiệu quả CPU, cung cấp swap hiệu quả mà không có hình phạt độ trễ của disk I/O:

```
# Cài đặt và cấu hình zram (Ubuntu/Debian)
sudo apt install zram-config

# Cấu hình thủ công để kiểm soát hơn
sudo modprobe zram

# Tạo thiết bị zram có kích thước 25% RAM (tốt cho server 32GB = 8GB zram)
echo "8G" | sudo tee /sys/block/zram0/disksize

# Dùng thuật toán lz4 (nén nhanh, tỷ lệ vừa phải)
echo "lz4" | sudo tee /sys/block/zram0/comp_algorithm

# Format và kích hoạt
sudo mkswap /sys/block/zram0
sudo swapon /sys/block/zram0 --priority 100

# Đặt swappiness thấp – chỉ swap dưới áp lực thực sự
echo "vm.swappiness=10" | sudo tee -a /etc/sysctl.conf
sudo sysctl -p
```

Với cấu hình này, server duy trì headroom cho các tiến trình agent ngay cả trong các Gradle build, và tỷ lệ nén zram (thường 2:1 đến 3:1 cho dữ liệu agent) cung cấp mở rộng bộ nhớ hiệu quả mà không có độ trễ disk I/O.

Tóm Tắt

Mở rộng quy mô hệ thống agent về cơ bản khác với mở rộng các web service không trạng thái vì:

1. **Các agent có trạng thái theo bản chất** — hiệu quả của chúng phụ thuộc vào context và bộ nhớ tích lũy, tạo ra các thách thức đồng bộ hóa khi bạn mở rộng ngang.

2. **Chi phí là siêu tuyến tính** — mỗi agent bạn thêm vào tạo ra các đường giao tiếp mới với các agent hiện có, và mỗi đường là một API call tiềm năng. Định cỡ model đúng và kiểm toán cron job tàn nhẫn không phải là tối ưu hóa tùy chọn; chúng là yêu cầu sống còn.
3. **Database HA không thể thương lượng, nhưng các chế độ thất bại là tinh vi** — cuối tuần 22 failover được gây ra bởi sự không khớp cấu hình vô hình dưới tải bình thường. Triển khai kiểm tra nhất quán cấu hình và chạy các cuộc diễn tập failover thường xuyên trước khi bạn cần chúng.
4. **Server ARM của bạn đáng tin cậy, không phải đa năng** — hồ sơ uptime không phải là sự chứng thực chung. Hãy khớp đặc tính phần cứng với đặc tính khối lượng công việc.
5. **Lượng tự động hóa phù hợp ít hơn bạn nghĩ** — các AI job tự động tạo ra giá trị tỷ lệ với khả năng hành động của chúng, không phải tần suất. Xóa các cron job mà không ai đọc.

Agent mesh bạn xây dựng ở tháng sáu sẽ trông không giống agent đơn lẻ bạn triển khai ở tháng một. Điều đó ổn. Kiến trúc nên phát triển khi bạn học được những gì các agent thực sự đang làm, chúng tốn bao nhiêu, và giá trị chúng trả về. Mở rộng có chủ đích, đo lường mọi thứ, và giữ lịch xác minh backup nhất quán hơn lịch triển khai của bạn.

Chương tiếp theo: Bài Học Rút Ra — những sai lầm tôi đã mắc để bạn không phải mắc.

Chương 16: Bài Học Rút Ra — Những Điều Tôi Ước Mình Biết Trước

“Bạn nghĩ 3 ngày tạo ra một ứng dụng triệu đô? 😄” — Một người dùng, sau khi tôi mô tả AI đã dành bao lâu để tạo code

Giới Thiệu

Mọi câu chuyện thực tế trong cuốn sách này đại diện cho một sai lầm thực sự. Một số tổn tiền. Một số tổn giấc ngủ. Một vài cái tổn niềm tin — với người dùng, với các bên liên quan, hoặc với chính tôi khi nhìn lại một tuần làm việc và nhận ra mình đã xây dựng đúng thứ sai.

Chương này khác với các chương khác. Không có sơ đồ kiến trúc. Không có file cấu hình ở nửa đầu. Chỉ là một bản kê khai trung thực về mười sai lầm có hậu quả lớn nhất tôi đã mắc khi xây dựng hệ thống AI agent trong môi trường thực tế, theo sau là suy ngẫm rộng hơn về yếu tố con người mà tài liệu kỹ thuật liên tục bỏ qua.

Tôi đã tổ chức chúng theo thứ tự gần đúng mà người thực hành mới sẽ gặp phải — có nghĩa là những sai lầm đầu tiên là vận hành, những cái ở giữa là kiến trúc, và những cái cuối cùng là triết học. Bạn có thể sẽ mắc một số sai lầm này dù sao, vì một số bài học chỉ đọng lại khi chúng tổn thứ gì đó từ bạn. Nhưng có lẽ một vài cái bạn có thể bỏ qua.

Top 10 Sai Lầm

Sai Lầm #1: Không Đặt Cảnh Báo Ngân Sách (Bài Học \$500)

Tháng đầu tiên tôi chạy agent trong môi trường thực tế mà không có cảnh báo ngân sách, tôi đã chi \$500 cho các API call mà tôi không có ý định thực hiện. Không phải vì các agent đang làm gì đó kịch tính — không có vòng lặp bùng phát, không có retry vô hạn. Chỉ là hoạt động bình thường, nhân với thực tế là tôi đã kết nối nhiều nguồn dữ liệu hơn tôi nhận ra, và các agent đang xử lý tất cả chúng.

Tôi nhận ra khi kiểm tra sao kê thẻ tín dụng. Không phải khi nó xảy ra. Ba tuần sau khi nó xảy ra.

Cách sửa là hai dòng cấu hình. Không có lý do gì để không làm điều này ngay từ ngày đầu.

Anthropic (qua AWS Bedrock):

```
# Đặt giới hạn chi tiêu cứng qua 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"
    }]
}]'

```

API Anthropic trực tiếp — tự tạo tracking:

```

# cost-tracker.py — bọc mọi 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}, # mỗi 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 # Cảnh báo ở 80% ngân sách

@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"Đã vượt ngân sách ngày ${DAILY_BUDGET_USD}. "
                f"Đã chi: ${self._today_spend:.2f}"
            )
        elif self._today_spend >= DAILY_BUDGET_USD * ALERT_THRESHOLD:

```

```

        self._send_alert(
            f"CẢNH BÁO: {self._today_spend/DAILY_BUDGET_USD*100:.0f}% "
            f"ngân sách ngày đã dùng (${self._today_spend:.2f}/${DAILY_BUDGET_USD})"
        )

    def _send_alert(self, message: str):
        # Cảnh báo Telegram – thay bằng kênh ưa thích của bạn
        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

```

Đặt ngân sách ngày trước khi bạn viết agent đầu tiên. Không phải sau khi bạn thấy hóa đơn.

Sai Lầm #2: Tin Tưởng Agent Với Cấu Hình Infrastructure (Sự Cố API Proxy)

Sớm trong dự án, tôi cho một agent quyền ghi vào các file cấu hình infrastructure. Nhiệm vụ của agent là tối ưu hóa cài đặt định tuyến API proxy service. Một buổi tối, trong khi tôi vắng mặt, nó quyết định “đơn giản hóa” cấu hình định tuyến API proxy service bằng cách xóa những gì nó xác định là các fallback route dư thừa.

Các fallback route không dư thừa. Chúng là các route phục vụ 40% lưu lượng khi các endpoint chính bị rate limit. Trong vòng sáu phút sau khi thay đổi, hệ thống đang trả về lỗi cho 40% yêu cầu. Agent không nhận ra — nó đã chuyển sang tác vụ tiếp theo rồi.

Quy tắc: Các agent nên đề xuất thay đổi infrastructure; con người nên áp dụng chúng. Đây không phải là sự hoài nghi về AI — đó là quản lý thay đổi đúng đắn.

```

# Mẫu: tạo đề xuất thay đổi, yêu cầu phê duyệt của con người
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"""
ĐỀ XUẤT THAY ĐỔI INFRASTRUCTURE
ID: {self.id}
Agent: {self.agent_id}
Loại: {self.change_type}

Lý do:
{self.rationale}

Diff:
{self.diff}

Trả lời APPROVE {self.id} hoặc REJECT {self.id}
"""

# Agent tạo đề xuất – KHÔNG áp dụng nó
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, tùy bạn
    # Agent dừng ở đây. Con người áp dụng sau khi review.
    return proposal

```

Infrastructure là lĩnh vực duy nhất mà chi phí của sai lầm agent là ngay lập tức, có thể nhìn thấy, và có thể ảnh hưởng đến khách hàng. Sự bất tiện của bước phê duyệt thủ công xứng đáng với điều đó.

Sai Lầm #3: Tự Phục Hồi Tốn Hơn Là Các Vấn Đề Nó Sửa

Tự phục hồi nghe có vẻ lý tưởng: một agent phát hiện vấn đề, chẩn đoán nó, sửa nó, và bạn thức dậy với một hệ thống khỏe mạnh. Thực tế phức tạp hơn.

Agent tự phục hồi đầu tiên của tôi được kích hoạt bởi mức sử dụng bộ nhớ cao trên Server-2. Cách sửa của nó là khởi động lại tiến trình nặng nhất. Tiến trình đó là một Cognie indexing job mất 45 phút để khởi động lại và khởi tạo lại. Agent kích hoạt khởi động lại, sau đó phát hiện bộ nhớ vẫn cao (quá trình khởi động lại đang tiến hành), và kích hoạt một lần khởi động lại khác. Bốn lần khởi động lại sau, indexing job đã chạy được 20 phút và gần hoàn thành, nhưng auto-healer đã kill nó bốn lần ở các giai đoạn khởi tạo ngày càng tốn kém.

Tổng chi phí của “cách sửa”: 3 giờ công việc indexing bị mất + 4 × API call Opus để chẩn đoán = khoảng \$8.40 trong token + ~3 giờ compute.

Vấn đề ban đầu: bộ nhớ ở 87%, nằm trong dải vận hành bình thường cho server đó trong quá trình indexing. Không cần can thiệp.

Quy tắc cho auto-healing agent: 1. Không bao giờ khởi động lại một tiến trình mà không kiểm tra xem nó có đang tiến triển không 2. Triển khai thời gian cooldown tối thiểu giữa các lần can thiệp (tôi dùng 30 phút) 3. Giới hạn can thiệp mỗi service mỗi ngày (tối đa 2) 4. Nếu cùng một cách sửa đã được áp dụng hai lần mà không có giải pháp, dừng lại và cảnh báo con người 5. Theo dõi chi phí của mỗi hành động auto-healing — nếu chi phí khắc phục trung bình > chi phí sự cố trung bình, vô hiệu hóa 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"Đã đạt tối đa can thiệp hàng ngày ({self.max_daily}) cho {service}"

        if today_interventions:
            last = max(today_interventions)
            if now - last < self.cooldown:
                remaining = self.cooldown - (now - last)
                return False, f"Cooldown đang hoạt động cho {service}: còn {remaining}"

        return True, "ok"

    def record_intervention(self, service: str):
        self._interventions[service].append(datetime.utcnow())
```

Sai Lầm #4: Không Báo Cáo Tiến Trình (Im Lặng Của Agent = Lo Lắng Người Dùng)

Các agent chậm. Một tác vụ phức tạp — phân tích codebase, tạo kế hoạch migration, review infrastructure — có thể mất 5 đến 15 phút. Trong thời gian đó, không có cập nhật tiến trình, người dùng giả định rằng có gì đó đã xảy ra sai. Họ gửi lại tác vụ. Bây giờ agent đang làm cùng một việc hai lần. Cả hai hoàn thành đều đến. Không ai biết cái nào là chuẩn.

Đây là vấn đề UX mà biến thành vấn đề tính đúng đắn.

```
# Wrapper báo cáo tiến trình
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 đã trôi qua"
        await self._send(msg)

    async def done(self, summary: str):
        elapsed = int(time.time() - self._start)
        await self._send(f"[{self.task_id}] HOÀN THÀNH ({elapsed}s): {summary}")

    async def error(self, message: str):
        elapsed = int(time.time() - self._start)
        await self._send(f"[{self.task_id}] LỖI ({elapsed}s): {message}")

    async def _send(self, message: str):
        # Triển khai phụ thuộc vào kênh của bạn (Telegram, Slack, v.v.)
        await telegram_send(self.channel, message)

# Sử dụng bên trong tác vụ agent chạy lâu
async def analyze_codebase(repo_path: str, reporter: ProgressReporter):
    await reporter.update("Đang quét cấu trúc repository", pct_complete=5)
    files = scan_repo(repo_path)

    await reporter.update(f"Tìm thấy {len(files)} file, bắt đầu phân tích", pct_complete=20)
    analysis = await analyze_files(files)

    await reporter.update("Đang tạo khuyến nghị", pct_complete=80)
    recommendations = await generate_recommendations(analysis)

    await reporter.done(f"Phân tích hoàn thành: {len(recommendations)} khuyến nghị")
    return recommendations
```

Tần suất cập nhật quan trọng: quá ít và người dùng lo lắng; quá nhiều và các cập nhật trở thành nhiễu. Một cập nhật mỗi 30-90 giây cho các tác vụ dài là baseline tốt.

Sai Lầm #5: Hardcode Credentials

Tôi đã hardcode credentials trong ba dự án agent riêng biệt trước khi học được bài học này. Mỗi lần, credential kết thúc trong git repository trong vòng 48 giờ — hoặc được commit trực tiếp hoặc được đưa vào debug log đã được commit. Mỗi lần, tôi xoay vòng key và dành hai giờ xác minh không có gì bị rò rỉ.

Cách sửa không phức tạp. Dùng HashiCorp Vault cho các secret xoay vòng, và biến môi trường (được quản lý bởi secrets manager, không phải file `.env` plaintext) cho mọi thứ khác.

```
# Cài đặt Vault cho agent credentials
# Lưu một secret
vault kv put secret/agents/anthropic \
  api_key="sk-ant-..." \
  rate_limit_tier="tier-3"

# Đọc trong tiến trình agent
export ANTHROPIC_API_KEY=$(vault kv get -field=api_key secret/agents/anthropic)

# Hoặc dùng Vault agent sidecar để tự động gia hạn:
# /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"
}
```

File template (`env.tpl`) tạo ra một file môi trường mà agent đọc khi khởi động. Vault tự động làm mới nó khi secret xoay vòng. Agent không bao giờ cần biết giá trị secret thực tế tại thời điểm triển khai.

Sai Lầm #6: Bỏ Qua Firewall (Server-3 UFW Chưa Bao Giờ Được Bật)

Server-3, server database và persistence của tôi, chạy được bốn tháng với UFW (Uncomplicated Firewall) chưa bao giờ được bật. Tôi biết điều này. Tôi cứ có ý định cài đặt nó. Server “nằm sau mạng riêng” và “không trực tiếp đối mặt với internet.” Đây chính xác là điều kiện mà bạn bị vi phạm chậm, qua lateral movement, thay vì nhanh, qua tấn công trực tiếp.

Không có gì xấu xảy ra. Tôi may mắn. Đừng dựa vào may mắn.

```
# Cấu hình UFW tối thiểu cho database server
# Chạy các lệnh này – đừng chỉ đọc chúng

# Cài đặt nếu chưa có
sudo apt install ufw

# Mặc định: từ chối tất cả incoming, cho phép tất cả outgoing
sudo ufw default deny incoming
sudo ufw default allow outgoing

# Cho phép SSH chỉ từ các IP quản lý đã biết
sudo ufw allow from 10.0.1.0/24 to any port 22

# Cho phép PostgreSQL chỉ từ các application server
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)

# Cho phép giao tiếp etcd peer chỉ giữa các node cluster
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

# Cho phép Patroni REST API trong cluster
sudo ufw allow from 10.0.1.0/24 to any port 8008

# Bật
sudo ufw --force enable
sudo ufw status verbose

# Xác minh
sudo ss -tlnp | grep -E "5432|2379|2380|8008"
```

Đừng chờ đến khi bạn hoàn thành các phần thú vị mới bật firewall. Bật nó vào ngày đầu tiên, trước khi bạn kết nối server với bất kỳ thứ quan trọng nào.

Sai Lầm #7: Trộn Dữ Liệu Agent (Vấn Đề Multi-Tenancy Cognition)

Điều này đã được đề cập trong Chương 15, nhưng nó xứng đáng được mở rộng như một sai lầm. Tôi chạy một instance Cognition chung cho nhiều agent phục vụ các dự án khách hàng khác nhau. Theo thời gian, knowledge graph tích lũy sự kiện từ tất cả các dự án.

Một agent đang làm việc trên Dự án Beta hỏi một câu hỏi về credentials database production. Cognee truy xuất một sự kiện nó đã học từ Dự án Alpha — vì cả hai dự án đều có các thực thể được gắn thẻ “production” và “database” và vector similarity đủ cao để bẻ mặt nó.

Agent không bao giờ exfiltrate credential. Nhưng nó đã dùng một địa chỉ database sai từ dự án sai để xây dựng một migration script, điều đó sẽ thất bại to tiếng khi chạy. Tôi đã phát hiện ra trong code review. Đó là một sự suýt sảy.

Cô lập tenant trong các hệ thống tri thức không phải tùy chọn khi bạn có nhiều khách hàng hoặc dự án. Chạy các instance Cognee riêng biệt, các database Neo4j riêng biệt, hoặc ít nhất duy trì sự tách biệt namespace nghiêm ngặt với lọc tại thời điểm truy vấn:

```
# Wrapper Cognee nhận biết tenant
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):
        # Tiên tố tất cả thực thể với 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:
        # Luôn lọc theo tenant
        results = await cognee.search(
            query,
            filters={"tenant_id": self.tenant_id}
        )
        # Kiểm tra kép: lọc ra bất kỳ kết quả nào không có tenant đúng
        return [r for r in results if r.metadata.get("tenant_id") == self.tenant_id]
```

Sai Lầm #8: Over-Engineering Monitoring (Bẫy Lặp v1→v2→v2.2→v2.3)

Tôi đã xây dựng một hệ thống monitoring agent mesh tùy chỉnh. Rồi xây dựng lại. Rồi vá hai lần. Quá trình:

- **v1:** Prometheus + Grafana đơn giản với các metrics agent cơ bản. Hoạt động tốt. Tôi quyết định nó không đủ toàn diện.
- **v2:** Thêm logic alerting tùy chỉnh, một Python service tương quan sự kiện agent, Loki để tổng hợp log, Tempo để distributed tracing. Mất hai tuần. Python service có bug. Các tương quan hầu hết sai.

- **v2.2:** Sửa các bug tương quan. Thêm nhiều dashboard hơn. Thêm định tuyến Alertmanager cho các mức độ nghiêm trọng khác nhau. Thêm Telegram alert formatter.
- **v2.3:** Nhận ra Tempo tracing không thực sự được dùng bởi ai. Xóa nó. Đơn giản hóa định tuyến Alertmanager. Kết thúc với thứ gì đó trông rất giống v1 với alerting tốt hơn một chút.

Tổng thời gian: khoảng sáu tuần trong ba tháng.

Monitoring thực sự quan trọng ở cuối: - Heartbeat agent (tiến trình agent có sống không?) - Chi phí token mỗi agent mỗi ngày (agent này có trong ngân sách không?) - Tỷ lệ lỗi mỗi agent (agent có thành công với tác vụ không?) - Độ sâu hàng đợi (có gì đang đơn ú không?) - Độ trễ replication database (replica có theo kịp không?)

Đó là năm metrics. Một cài đặt Prometheus cơ bản nắm bắt tất cả. Correlation engine phức tạp không cần thiết vì khi agent thất bại, chúng thất bại rõ ràng — hoặc chúng ngừng phản hồi, hoặc chúng bắt đầu tạo ra lỗi, hoặc chi phí tăng vọt. Bạn không cần tương quan tinh vi để nhận ra bất kỳ điều nào trong số đó.

Xây dựng monitoring đơn giản trước. Thêm độ phức tạp chỉ khi bạn có câu hỏi cụ thể mà monitoring đơn giản không thể trả lời.

Sai Lầm #9: Dùng Model Sai Cho Tác Vụ Sai (Gemini Cho Infrastructure = Thất Bại)

Không phải mọi AI model đều tốt ngang nhau cho mọi tác vụ. Điều này có vẻ hiển nhiên nhìn lại. Nó không hiển nhiên khi tôi đang ở giữa một sprint tối ưu hóa chi phí và quyết định định tuyến các tác vụ quản lý infrastructure đến Gemini 2.5 Flash vì nó rẻ hơn Claude Sonnet.

Gemini 2.5 Flash là một model xuất sắc cho nhiều tác vụ. Tạo cấu hình infrastructure không phải một trong số đó — ít nhất không phải cho trường hợp dùng cụ thể của tôi là tạo Ansible playbook, cấu hình Patroni, và systemd unit file cho một stack cụ thể.

Các output hợp lệ về mặt cú pháp nhưng sai về mặt hoạt động. Các tác vụ Ansible sẽ thực thi mà không có lỗi nhưng không đạt được trạng thái dự kiến. Các systemd unit file với cú pháp đúng nhưng thứ tự phụ thuộc sai. Các cấu hình Patroni hợp lý nhưng có các tương tác tham số tinh vi gây ra vấn đề dưới tải.

Vấn đề không phải là khả năng — Gemini 2.5 Flash là model có khả năng. Vấn đề là stack cụ thể của tôi (Traefik + Patroni + etcd + IPC agent tùy chỉnh) đủ bất thường để phân phối training của model không bao gồm nhiều về nó. Claude Sonnet, có thể do dữ liệu training khác nhau hoặc nhấn mạnh RLHF, hoạt động đáng chú ý tốt hơn trên sự kết hợp cụ thể này.

```
# Lựa chọn model nên cụ thể theo loại TÁC VỤ VÀ lĩnh vực
MODEL_ROUTING = {
    # (task_type, domain): model
    ("generate", "infrastructure"): "claude-sonnet-4-6", # Tốt hơn trên stack đặc biệt
    ("generate", "python"): "claude-sonnet-4-6", # Cái nào cũng được
    ("generate", "javascript"): "gpt-4o", # Mạnh về hệ sinh thái JS
    ("analyze", "logs"): "gemini-2.5-flash", # Tuyệt vời về khớp mẫu
    ("analyze", "metrics"): "gemini-2.5-flash", # Nhanh và rẻ
    ("classify", "alerts"): "claude-haiku-4-5", # Nhanh, rẻ, đáng tin cậy
    ("summarize", "documents"): "gemini-2.5-pro", # Xuất sắc về context dài
    ("reason", "complex"): "claude-opus-4-6", # Tốt nhất cho vấn đề mới
}

def route_to_model(task_type: str, domain: str) -> str:
    key = (task_type, domain)
    if key in MODEL_ROUTING:
        return MODEL_ROUTING[key]
    # Mặc định
    return "claude-sonnet-4-6"
```

Bài học không phải là “dùng một model cho mọi thứ” — điều đó vừa tốn kém vừa thường sai. Bài học là: kiểm tra các tác vụ cụ thể của bạn với các model cụ thể của bạn trước khi cam kết với một quyết định tuyển. Những gì hoạt động tốt trong benchmark chung có thể không hoạt động tốt cho sự kết hợp yêu cầu cụ thể của bạn.

Sai Lầm #10: Không Kiểm Tra Output Sub-Agent (Thất Bại Thầm Lặng)

Khi orchestrator ủy thác tác vụ cho sub-agent và sub-agent trả về output, orchestrator thường tin tưởng output đó. Điều này sai.

Sub-agent thất bại thâm lặng theo những cách không phải lúc nào cũng rõ ràng: - Chúng trả về output trông hợp lý nhưng không đúng - Chúng ảo giác đường dẫn file, tên function, hoặc giá trị cấu hình - Chúng xác định đúng vấn đề nhưng đề xuất cách sửa giải quyết triệu chứng thay vì nguyên nhân - Chúng trả về câu trả lời đúng cho một phiên bản khác của phần mềm

Orchestrator, nhận output này, tiến hành như thể nó đúng. Lỗi lan truyền downstream. Đến khi nó bề mặt — trong một lần triển khai thất bại, một service cấu hình sai, hoặc lỗi runtime — output của sub-agent cách ba bước khởi điểm thất bại và khó truy ngược lại.

```
# Framework xác thực output cho phần hồi sub-agent
from typing import Protocol, TypeVar
import jsonschema

T = TypeVar("T")

class OutputValidator(Protocol[T]):
    def validate(self, output: str) -> T:
        """Phân tích và xác thực output. Raise ValidationError nếu không hợp lệ."""
        ...

class AnsiblePlaybookValidator:
    REQUIRED_KEYS = {"name", "hosts", "tasks"}

    def validate(self, output: str) -> dict:
        # Bước 1: Phân tích YAML
        try:
            parsed = yaml.safe_load(output)
        except yaml.YAMLError as e:
            raise ValidationError(f"YAML không hợp lệ: {e}")

        # Bước 2: Xác thực schema
        if not isinstance(parsed, list):
            raise ValidationError("Playbook phải là danh sách các play")

        for play in parsed:
            missing = self.REQUIRED_KEYS - set(play.keys())
            if missing:
                raise ValidationError(f"Play thiếu các key bắt buộc: {missing}")

        # Bước 3: Kiểm tra dry-run (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"Kiểm tra cú pháp thất bại: {result.stderr.decode()}")

        return parsed

# Trong 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:
    # Yêu cầu sub-agent sửa output
    fix_request = f"""
Output trước đó của bạn thất bại xác thực:
Lỗi: {e}

Output ban đầu:
{raw_output}

Vui lòng sửa output và chỉ trả về phiên bản đã sửa.
"""
    corrected = await sub_agent.execute(fix_request)
    return validator.validate(corrected) # Raise nếu vẫn không hợp lệ

```

Xây dựng validator cho mọi loại output sub-agent quan trọng. Khoản đầu tư hoàn vốn ngay lập tức trong lần đầu tiên sub-agent trả về rác hợp lý và orchestrator từ chối nó thay vì hành động theo nó.

Yếu Tố Con Người

AI Tạo Nhanh; Mọi Thứ Khác Mất Lâu Gấp 10 Lần

Có một ảo tưởng dai dẳng trong không gian AI tooling rằng các AI agent sẽ nén đáng kể timeline để xây dựng phần mềm. Ảo tưởng này có một sự thật hẹp trong đó: các AI agent thực sự tạo code, kế hoạch, và tài liệu nhanh hơn đáng kể so với con người làm việc một mình.

Ảo tưởng thất bại vì tạo code không phải nút thắt cổ chai.

Một người dùng đã nói với tôi, sau khi tôi mô tả cách agent đã sản xuất triển khai tính năng hoàn chỉnh trong ba ngày: “Bạn nghĩ 3 ngày tạo ra một ứng dụng triệu đô?” Câu trả lời, rõ ràng, là không. Ba ngày tạo AI được theo sau bởi:

- Hai tuần kiểm thử tích hợp (code được tạo hoạt động độc lập; tích hợp nó với hệ thống hiện có đòi hỏi làm lại đáng kể)
- Một tuần tối ưu hóa hiệu suất (triển khai được tạo đúng nhưng chậm; nó thực hiện các truy vấn database N+1 và không thể xử lý tải production)
- Ba ngày security review (AI đã triển khai đúng happy path và hoàn toàn bỏ qua input validation trên một số endpoint)
- Bốn ngày đánh bóng (UX hoạt động nhưng không tốt; người dùng thực đã thử và thấy nó khó hiểu theo những cách AI không lường trước)

- Một tuần monitoring production và xử lý sự cố (bốn vấn đề sau triển khai, ba trong số đó AI không mô hình hóa)

Timeline thực tế cho “3 ngày làm việc AI”: khoảng 7 tuần.

Đây không phải lập luận chống lại việc dùng AI agent. Tính năng có thể mất 12-16 tuần mà không có sự trợ giúp của AI. Đòn bẩy là thực. Nhưng đòn bẩy nằm ở giai đoạn tạo, chỉ là một phần của việc giao hàng phần mềm. Các giai đoạn build, test, integrate, polish, và operate vẫn mất khoảng thời gian như chúng đã luôn mất.

Hãy lên kế hoạch phù hợp. Đừng cam kết với timeline dựa trên tốc độ AI có thể viết code. Cam kết với timeline dựa trên thời gian toàn bộ chu kỳ giao hàng mất — với AI tăng tốc giai đoạn tạo và không nhiều thứ khác.

Vấn Đề Hiệu Chỉnh Tin Tưởng

Làm việc với AI agent hàng ngày tạo ra vấn đề hiệu chỉnh. Bạn thấy agent thành công với các tác vụ ấn tượng và bắt đầu mở rộng tin tưởng đến các lĩnh vực nơi agent kém đáng tin cậy hơn. Bạn ngừng kiểm tra output mà agent thường làm đúng, có nghĩa là bạn ngừng bắt được output sai thỉnh thoảng trong danh mục đó.

Điều ngược lại cũng xảy ra: agent mắc lỗi rõ ràng, bạn mất niềm tin vào nó, và ngừng ủy thác các tác vụ nó xử lý tốt.

Duy trì hiệu chỉnh chính xác — không tin tưởng quá mức cũng không tin tưởng thiếu — đòi hỏi ghi log output có hệ thống và review định kỳ. Tôi làm spot-check hàng tuần: chọn 20 output agent ngẫu nhiên từ tuần, xác minh thủ công, ghi chú bất kỳ cái nào sai. Điều này mất khoảng 45 phút và giữ hiệu chỉnh của tôi trung thực.

Dự Đoán Cho Lĩnh Vực

Tôi đang viết điều này vào tháng 3 năm 2026. Phần sau phản ánh nơi tôi kỳ vọng lĩnh vực di chuyển trong 12-18 tháng tới dựa trên các quỹ đạo hiện tại. Hãy coi đây là suy đoán có căn cứ, không phải lời tiên tri.

Tiêu Chuẩn Hóa MCP Sẽ Trở Thành Table Stakes

Model Context Protocol (MCP) đã nổi lên như tiêu chuẩn thống trị để kết nối AI agent với các công cụ và nguồn dữ liệu bên ngoài. Đầu năm 2025, tích hợp một agent với nguồn dữ liệu mới có nghĩa là viết code tùy chỉnh mỗi lần. Cuối năm 2025, hầu hết các framework agent nghiêm túc đã có hỗ trợ MCP. Đến giữa năm 2026, tôi kỳ vọng tuân thủ MCP sẽ là yêu cầu cứng cho bất kỳ tích hợp agent cấp enterprise nào.

Về mặt vận hành, điều này quan trọng vì nó thay đổi gánh nặng bảo trì. Thay vì duy trì các tích hợp riêng với mười lăm nguồn dữ liệu khác nhau, bạn duy trì một MCP adapter và để các framework xử lý phần còn lại.

Context Window 2M+ Sẽ Thay Đổi Kiến Trúc Bộ Nhớ

Context window đang mở rộng nhanh chóng: 200K → 1M → và có thể 2M+ token vào cuối năm 2026. Điều này sẽ thay đổi kinh tế của kiến trúc bộ nhớ phân tầng được mô tả trong Chương 15.

Khi bạn có thể đặt toàn bộ codebase, sáu tháng lịch sử sự cố gần đây, và một knowledge graph hoàn chỉnh trong một context window duy nhất, trường hợp cho các hệ thống bộ nhớ ngoài phức tạp yếu đi. Độ trễ và chi phí của các truy vấn Cognee graph (20-200ms mỗi lần tra cứu) trở nên khó biện minh hơn khi bạn có thể đạt được cùng khả năng hồi tưởng bằng cách đơn giản là nhồi thêm vào context.

Các agent xử lý tốt quá trình chuyển đổi này sẽ là những cái được xây dựng trên các abstraction truy xuất — nơi chi tiết triển khai “truy xuất từ Cognee” vs “truy xuất từ context” được ẩn sau một interface sạch có thể chuyển đổi triển khai khi cấu trúc chi phí thay đổi.

```
# Xây dựng abstraction truy xuất bây giờ, tối ưu triển khai sau
class AgentMemory(Protocol):
    async def store(self, key: str, content: str, **metadata) -> None: ...
    async def retrieve(self, query: str, limit: int = 5) -> list[MemoryItem]: ...

# Hôm nay: truy xuất Cognee graph (cần thiết cho context window 200K)
class CogneeMemory:
    async def retrieve(self, query: str, limit: int = 5) -> list[MemoryItem]:
        return await cognee.search(query, top_k=limit)

# Ngày mai: truy xuất context-window (khi window mở rộng đến 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]:
```

```
# Với 2M context, chỉ bao gồm mọi thứ liên quan
return self._semantic_filter(self._context, query, limit)
```

Suy Luận Thích Nghi Sẽ Thay Thế Prompting Tĩnh

Phương pháp thống trị hiện tại — viết một system prompt chi tiết và hy vọng nó bao gồm đủ trường hợp — dễ vỡ. Các agent thất bại ở các edge case không được dự đoán trong prompt. Thêm hướng dẫn vào prompt làm nó dài hơn, điều này làm giảm hiệu suất trên các trường hợp thông thường.

Giải pháp thay thế đang nổi lên là suy luận thích nghi: các agent điều chỉnh cách tiếp cận của họ dựa trên những gì họ quan sát trong quá trình thực thi tác vụ, thay vì tuân theo một tập hướng dẫn tĩnh. Các model với khả năng suy luận mạnh hơn (Claude Opus 4.6 là benchmark hiện tại, với bất cứ điều gì theo sau nó trong 2026-2027) đang làm điều này thực tế.

Về mặt vận hành, suy luận thích nghi thay đổi cách bạn viết hướng dẫn agent: ít đặc tả các bước chính xác hơn, nhiều đặc tả mục tiêu và ràng buộc hơn. Agent tự tìm ra các bước. Điều này khó debug hơn khi nó sai — “agent đã làm gì đó bất ngờ” là chế độ thất bại phức tạp hơn “agent đã làm bước 7 không đúng.” Hãy lên kế hoạch đầu tư nhiều hơn vào tracing và ghi log output khi suy luận thích nghi trở nên phổ biến.

Lời Khuyên Cuối Cho DevOps Engineers Bước Vào Không Gian AI Agent

Sau hơn hai năm xây dựng, mở rộng, phá vỡ, và xây dựng lại các hệ thống AI agent trong môi trường thực tế, đây là những gì tôi sẽ nói với ai đó bắt đầu hôm nay:

Bắt đầu với một agent, một tác vụ, một thước đo thành công. Sự cám dỗ để xây dựng mesh ngay lập tức rất mạnh. Chống lại nó. Một agent với một công việc rõ ràng và một metric thành công rõ ràng dạy cho bạn nhiều hơn về hành vi agent so với mười agent làm công việc được định nghĩa lỏng lẻo.

Đo lường mọi thứ từ ngày đầu tiên. Số token, chi phí, độ trễ, tỷ lệ lỗi, chất lượng output. Không phải vì bạn sẽ đọc tất cả các dashboard — bạn sẽ không — mà vì khi có gì đó sai ở tháng thứ sáu, bạn sẽ cần dữ liệu lịch sử để hiểu khi nào nó bắt đầu.

Đối xử với output agent như code từ một developer junior. Không phải với sự nghi ngờ, mà với review phù hợp. Xác minh vài output đầu tiên trong bất kỳ lĩnh vực khả năng mới nào. Spot-check thường xuyên. Xây dựng validator tự động cho các loại output có stakes cao.

Bản năng DevOps của bạn có thể chuyển giao hơn bạn nghĩ. Các nguyên tắc tạo nên infrastructure đáng tin cậy — dư thừa, quan sát, graceful degradation, kiểm soát thay đổi, least privilege — áp dụng trực tiếp cho các hệ thống agent. Chi tiết triển khai khác. Các nguyên tắc giống nhau.

Lập ngân sách cho bất ngờ. Cả về tiền và thời gian. Các hệ thống agent làm những gì bạn không dự đoán — đôi khi hữu ích, đôi khi tốn kém, đôi khi cả hai. Giữ capacity dự trữ trong ngân sách và lịch trình cho những điều bất ngờ.

Các agent không phải là sản phẩm. Các kết quả chúng cho phép mới là sản phẩm. Một agent tốn \$10/ngày và tiết kiệm 4 giờ làm việc của con người mỗi ngày là tốt. Một agent tốn \$10/ngày và tạo ra output mà không ai đọc không phải tự động hóa — đó là sân khấu đắt tiền. Đánh giá các agent dựa trên giá trị chúng tạo ra, không phải về ấn tượng của output.

Lĩnh vực đang di chuyển nhanh. Các công cụ, giá, và kiến trúc cụ thể trong cuốn sách này sẽ lão hóa. Phán đoán vận hành để biết khi nào thứ gì đó đang hoạt động, khi nào nó không, và phải làm gì — điều đó lão hóa chậm hơn. Xây dựng phán đoán đó một cách có chủ đích, từ agent đầu tiên bạn triển khai.

Kết thúc Phần 5.

Các phụ lục tiếp theo chứa tài liệu tham khảo: tóm tắt công cụ và framework (Phụ lục A), giá AI model tính đến tháng 3 năm 2026 (Phụ lục B), và danh sách kiểm tra production bạn thực sự nên dùng (Phụ lục C).

PHỤ LỤC

Phụ Lục A: Tài Liệu Tham Khảo Công Cụ và Framework

Cẩm nang thực địa về các công cụ xuất hiện xuyên suốt cuốn sách này — mỗi công cụ làm gì, khi nào nên dùng, và những gì sẽ cần bạn nếu bạn không cần thận.

Phụ lục này được tổ chức theo danh mục. Với mỗi công cụ, bạn sẽ tìm thấy: mô tả bằng ngôn ngữ đơn giản, các trường hợp dùng mà nó tỏa sáng, và những gotcha mà kinh nghiệm thực tế tiết lộ nhưng tài liệu hiếm khi đề cập. Phiên bản và giá tính đến tháng 3 năm 2026.

Framework Agent

CrewAI

Làm gì: Điều phối nhiều AI agent trong các workflow dựa trên vai trò. Bạn định nghĩa các agent với vai trò cụ thể (researcher, writer, engineer), giao cho họ tác vụ, và CrewAI quản lý luồng thực thi — tuần tự, song song, hoặc phân cấp.

Khi nào dùng: - Pipeline đa agent nơi các agent khác nhau có lĩnh vực chuyên môn riêng - Workflow ánh xạ tự nhiên đến cấu trúc nhóm con người - Prototyping nhanh các mẫu cộng tác agent

Các gotcha quan trọng: - Bộ nhớ giữa các agent trong một crew được chia sẻ theo mặc định — cô lập tenant đòi hỏi cấu hình rõ ràng - Chế độ `process=Process.hierarchical` thêm một lần gọi manager LLM cho mỗi ủy thác tác vụ; chi phí tích lũy nhanh hơn dự kiến trong các crew lớn - Định nghĩa công cụ dài dòng và lặp lại; xây dựng tool registry để tránh copy-paste - Chất lượng output crew giảm đáng kể khi crew vượt quá 5-6 agent; context bị pha loãng quá nhiều

```
from crewai import Agent, Task, Crew, Process
```

```
infra_agent = Agent(
```

```

    role="Infrastructure Engineer",
    goal="Phân tích và tối ưu hóa cấu hình server",
    backstory="Chuyên gia về Linux systems, Patroni, và container orchestration",
    tools=[bash_tool, file_reader_tool],
    llm="claude-sonnet-4-6",
    max_iter=5,          # Ngăn vòng lặp bùng phát
    memory=True,
)

analysis_task = Task(
    description="Phân tích cấu hình PostgreSQL tại /etc/postgresql/15/main/postgresql.conf",
    expected_output="Danh sách các cài đặt không tối ưu với giá trị được khuyến nghị và lý do",
    agent=infra_agent,
)

crew = Crew(
    agents=[infra_agent],
    tasks=[analysis_task],
    process=Process.sequential,
    verbose=True,
)

```

LangChain

Làm gì: Framework để kết hợp các lệnh gọi LLM, công cụ, bộ nhớ, và chuỗi thành ứng dụng. Giống toolkit hơn là framework có quan điểm — cung cấp cho bạn các primitive để xây dựng hơn là kiến trúc cố định.

Khi nào dùng: - Ứng dụng cần kiểm soát chi tiết các tương tác LLM - Pipeline Retrieval-Augmented Generation (RAG) - Khi bạn cần kết hợp nhiều LLM provider trong một ứng dụng

Các gotcha quan trọng: - Lớp abstraction thêm độ trễ đáng kể (50-200ms mỗi bước chuỗi) — đo trước khi giả định nó chấp nhận được - Các breaking change thường xuyên giữa các minor version; pin version nghiêm ngặt và kiểm tra upgrade rõ ràng - LCEL (LangChain Expression Language) mạnh mẽ nhưng tạo ra stack trace gần như không thể đọc khi thứ gì đó thất bại sâu trong chuỗi - Tích hợp observability LangSmith xuất sắc nhưng yêu cầu tài khoản trả phí để dùng production; hãy lập ngân sách cho nó

AutoGen (Microsoft)

Làm gì: Framework hội thoại đa agent nơi các agent giao tiếp qua truyền tin có cấu trúc. Đặc biệt mạnh trong các workflow tạo code nơi các agent viết code, thực thi, quan sát kết quả, và lặp lại.

Khi nào dùng: - Vòng lặp tạo và thực thi code - Workflow hưởng lợi từ tranh luận giữa agent (một agent đề xuất, agent khác phê bình) - Tích hợp với Azure OpenAI (hỗ trợ hạng nhất)

Các gotcha quan trọng: - Kết thúc hội thoại đòi hỏi cấu hình rõ ràng; nếu không có, các agent sẽ tiếp tục hội thoại vô thời hạn (và tính phí vô thời hạn) - Môi trường thực thi code chạy cục bộ theo mặc định; sandbox nó trước khi chạy trong production - Lịch sử tin nhắn tăng theo mỗi lần trao đổi — đặt `max_consecutive_auto_reply` để ngăn tràn context window

```
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="Bạn là một infrastructure engineer. Viết và kiểm tra Ansible playbook.",
    max_consecutive_auto_reply=5, # QUAN TRỌNG: Luôn đặt cái này
)

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

Làm gì: AI coding assistant của Anthropic, được thiết kế để làm việc trong môi trường phát triển. Đọc codebase, viết và chỉnh sửa file, chạy lệnh, và hoạt động qua vòng lặp tool-use với giám sát của con người.

Khi nào dùng: - Tác vụ phát triển phần mềm: viết, review, refactor, debug - Khám phá repository và tạo tài liệu - Tác vụ đòi hỏi chỉnh sửa file lặp lại với hiểu biết về toàn bộ codebase

Các gotcha quan trọng: - Claude Code hoạt động với quyền file của người dùng đã khởi chạy nó — chạy với người dùng có đặc quyền thấp chuyên dụng trong các môi trường gần production - Các phiên dài tích lũy context lớn; khởi động lại phiên định kỳ cho các tác vụ kéo dài nhiều giờ - Vòng lặp tool-use có thể bị kẹt trong “tê liệt phân tích” với các tác vụ mơ hồ; cung cấp tiêu chí thành công rõ ràng

Gateway Agent (Orchestrator)

Làm gì: AI gateway agent được thiết kế để bổ sung cho Claude Code. Xử lý các tác vụ mà coding agent không thể: duyệt web, tự động hóa browser, nhắn tin Telegram, tác vụ theo lịch, và kiểm soát thiết bị (camera, màn hình, vị trí trên thiết bị di động được ghép cặp).

Khi nào dùng: - Tìm kiếm web và truy xuất dữ liệu trực tiếp - Gửi thông báo qua Telegram, WhatsApp, hoặc Discord - Tự động hóa browser (điền form, scrape các trang được xác thực) - Lên lịch nhắc nhở và tác vụ định kỳ - Điều phối với Claude Code qua shared inbox tại `~/ .agent-gateway/workspace/`

Các gotcha quan trọng: - Nhắn tin liên agent qua shared inbox dựa trên file; cả hai agent phải đang chạy để giao hàng đáng tin cậy - Kết quả duyệt web bao gồm nội dung trang đầy đủ — cắt tỉa tích cực trước khi đưa vào LLM prompt hoặc chi phí tăng vọt - Kiểm soát thiết bị di động yêu cầu app companion gateway và sự đồng ý rõ ràng của người dùng mỗi phiên

```
# Gửi tác vụ đến gateway agent
gateway system event --text "Tìm kiếm ghi chú phát hành Patroni 4.x và tóm tắt các breaking change"

# Kiểm tra inbox để nhận phản hồi của gateway agent
cat ~/ .agent-gateway/workspace/inbox/from-gateway.md
```

Infrastructure

Docker

Làm gì: Container hóa các ứng dụng và dependencies của chúng thành các image portable, có thể tái tạo. Nền tảng của triển khai nhất quán trên các môi trường development, staging, và production.

Khi nào dùng: Hầu như luôn luôn. Nếu bạn không container hóa các agent service, bạn đang tạo ra các vấn đề không nhất quán môi trường sẽ bề mặt vào thời điểm tệ nhất có thể.

Các gotcha quan trọng: - Giới hạn bộ nhớ container tương tác kém với GC của Python; đặt `--memory-swap` bằng `--memory` để ngăn swap thrashing - Các tiến trình agent tạo ra subprocess (để thực thi code, tool use) có thể thoát

khởi giới hạn tài nguyên container qua `fork()` trừ khi bạn dùng `--pids-limit` - Các lớp image cho AI Python dependencies lớn (2-5GB là phổ biến); dùng multi-stage build để giảm kích thước image production

```
# Multi-stage build cho 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
# Không bao giờ chạy AI agent với quyền root
RUN useradd -m -u 1000 agent
USER agent
CMD ["python", "-m", "agent.main"]
```

Traefik

Làm gì: Reverse proxy và load balancer cloud-native với tính năng tự động service discovery qua Docker labels, Kubernetes, Consul, và etcd. Xử lý TLS termination, định tuyến, và middleware.

Khi nào dùng: - HTTP routing cho agent REST API - TLS tự động qua Let's Encrypt - Traffic splitting và weighted load balancing trên các instance agent

Các gotcha quan trọng: - Dashboard hiển thị cấu hình định tuyến theo mặc định trên port 8080 — hạn chế truy cập ngay lập tức - Thứ tự middleware quan trọng; `rateLimit` trước `retry` ngăn các retry bypass giới hạn tốc độ - Khoảng kiểm tra sức khỏe quá tích cực (< 5s) sẽ tạo ra nhiều log đáng kể và overhead nhỏ

HAProxy

Làm gì: TCP/HTTP load balancer và proxy được kiểm chứng kỹ. Ít tính năng hơn Traefik cho môi trường động, nhưng hiệu suất có thể dự đoán hơn dưới tải cực cao và hỗ trợ tốt hơn cho các giao thức không phải HTTP.

Khi nào dùng: - Front-end connection pooling PostgreSQL (trước PgBouncer) - TCP load balancing cho giao tiếp agent không phải HTTP - Các tình huống cần quyết định định tuyến dưới mili giây với độ trễ có thể dự đoán

Các gotcha quan trọng: - Cấu hình HAProxy yêu cầu khởi động lại hoàn toàn cho hầu hết các thay đổi (không có hot reload cho các thay đổi backend trong các phiên bản cũ hơn) - Trang thống kê nên được bảo vệ với basic auth tối thiểu; nó tiết lộ số lượng kết nối và sức khỏe server - Session stickiness (balance source) có thể gây phân phối tải không đều khi một số lượng nhỏ IP tạo ra hầu hết lưu lượng

Patroni

Làm gì: High-availability manager cho PostgreSQL. Dùng hệ thống đồng thuận phân tán (etcd, Consul, hoặc ZooKeeper) để quản lý leader election, automatic failover, và thăng cấp replica.

Khi nào dùng: - Bất kỳ triển khai PostgreSQL production nào phải sống sót qua server failure - Cluster PostgreSQL đa server

Các gotcha quan trọng: Xem Chương 15 để biết toàn bộ postmortem. Phiên bản ngắn: - Mỗi node phải có cấu hình etcd peer giống hệt nhau — configuration drift gây ra failover giả - Đặt `maximum_lag_on_failover` để ngăn thăng cấp replica tụt hậu đáng kể; 1MB là điểm khởi đầu hợp lý - `use_pg_rewind` phải được bật trước failover, không phải sau — bật nó tại thời điểm tạo cluster - Luôn dùng Patroni trước khi dùng etcd; dùng etcd trước có thể kích hoạt failover không cần thiết

etcd

Làm gì: Kho lưu trữ key-value phân tán cung cấp đảm bảo nhất quán mạnh qua thuật toán đồng thuận Raft. Được dùng bởi Patroni để bầu chọn leader và trạng thái cluster.

Khi nào dùng: Khi bạn cần đồng thuận phân tán để bầu chọn leader, quản lý cấu hình, hoặc service discovery.

Các gotcha quan trọng: - Tối thiểu ba node để có fault tolerance có ý nghĩa; hai node không cung cấp bảo vệ (cả hai thất bại đồng thời trên bất kỳ phân vùng nào) - etcd nhạy cảm với độ trễ disk — chạy nó trên SSD; etcd được hỗ trợ bởi HDD sẽ tạo ra election timeout dưới tải - Bật auto-compaction (`auto-compaction-retention`) hoặc thư mục dữ liệu sẽ tăng vô tận và cuối cùng gây OOM - Giá trị `initial-cluster` phải giống hệt nhau trên tất cả node; không khớp gây ra các triệu chứng split-brain sau khi khởi động lại

Monitoring

Prometheus

Làm gì: Thu thập và lưu trữ time-series metrics. Scrape các endpoint metrics theo khoảng thời gian có thể cấu hình, lưu trong TSDB cục bộ, và cung cấp ngôn ngữ truy vấn (PromQL) để phân tích và alerting.

Khi nào dùng: Luôn luôn. Nếu bạn có service production, nó cần Prometheus metrics.

Các gotcha quan trọng: - Retention mặc định là 15 ngày; để phân tích xu hướng bạn cần retention dài hơn hoặc storage backend từ xa (Thanos, Cortex) - Label cardinality cao (agent ID, user ID, request ID) gây ra TSDB bloat — kiểm soát label cardinality - Bản thân Prometheus server là điểm lỗi đơn; chạy hai instance scrape cùng target để giám sát các màn hình

```
# Các metrics agent quan trọng để expose
# Trong endpoint metrics của agent:

# Tiêu thụ token
agent_tokens_total{model="claude-sonnet-4-6", type="input"} 1523000
agent_tokens_total{model="claude-sonnet-4-6", type="output"} 284000

# Kết quả tác vụ
agent_tasks_total{status="success"} 842
agent_tasks_total{status="error"} 17
agent_tasks_total{status="timeout"} 3

# Histogram độ trễ
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

# Theo dõi chi phí
agent_cost_usd_total{agent_id="claude-code-1"} 47.23
```

Grafana

Làm gì: Nền tảng visualization và dashboard. Kết nối với Prometheus, Loki, Tempo, và hàng chục nguồn dữ liệu khác để tạo dashboard vận hành và alert.

Khi nào dùng: Cùng với Prometheus. Hai cái này thực sự là một đơn vị.

Các gotcha quan trọng: - Cấu hình dashboard chỉ được lưu trong database SQLite của Grafana không được backup theo mặc định — export dashboard dưới dạng JSON và commit vào git - Đánh giá alert xảy ra trong Grafana

nhưng định tuyến thông báo xảy ra trong Alertmanager; hiểu hệ thống nào chịu trách nhiệm về cái gì trước khi debug một alert bị bỏ qua - Credentials mặc định admin/admin thực sự được người ta dùng trong production; thay đổi chúng trong lần đăng nhập đầu tiên

Alertmanager

Làm gì: Xử lý định tuyến, deduplication, grouping, và silencing các alert từ Prometheus. Gửi thông báo đến Telegram, Slack, PagerDuty, email, và nhiều kênh khác.

Khi nào dùng: Cùng với Prometheus để alerting production.

Các gotcha quan trọng: - Định tuyến alert là cấu trúc cây — một alert khớp route đầu tiên và dừng lại (trừ khi có `continue: true`); các bug logic định tuyến khiến alert đến sai kênh một cách thảm lạng - Các tham số `group_wait` và `group_interval` xác định Alertmanager chờ bao lâu trước khi gửi alert được nhóm; mặc định thường quá dài để phản hồi sự cố - Silence một instance alert cụ thể (dùng matchers), không phải toàn bộ tên alert — silencing theo tên tắt tiếng tất cả instance bao gồm các instance mới kích hoạt vì lý do khác

Loki

Làm gì: Hệ thống tổng hợp log được thiết kế để làm việc cùng với Prometheus. Dùng cùng mô hình label với Prometheus cho log stream, làm cho tương quan log-metric tự nhiên trong Grafana.

Khi nào dùng: - Lưu trữ log tập trung và tìm kiếm trên nhiều agent server - Tương quan log với metrics trong cùng Grafana dashboard

Các gotcha quan trọng: - Loki được tối ưu cho log stream với label low-cardinality — không đặt các giá trị động (request ID, nội dung được tạo bởi agent) trong label - Giới hạn tốc độ nhận log sẽ thảm lạng drop log nếu bị vượt; cấu hình `limits_config.ingestion_rate_mb` phù hợp với volume log của bạn - Ngôn ngữ truy vấn `logql` mạnh mẽ nhưng full-text search (`|= "error"`) là sequential scan — dùng label selector để thu hẹp không gian tìm kiếm trước

Tempo

Làm gì: Backend distributed tracing. Lưu trữ và truy vấn trace từ các ứng dụng được đo lường với OpenTelemetry.

Khi nào dùng: Khi bạn có workflow đa agent cần trace một request qua nhiều agent hop và hiểu thời gian được dùng ở đâu.

Các gotcha quan trọng: - Trace chỉ hữu ích nếu mọi thành phần trong đường dẫn được đo lường — đo lường một phần tạo ra trace không đầy đủ gây hiểu nhầm - Chi phí lưu trữ tăng tỷ lệ với volume request và độ sâu trace; sample tích cực (1-10% trace cho các đường dẫn volume cao) - Như đã lưu ý trong Chương 16, Tempo dễ cài đặt và dễ ngừng dùng vì không ai nhìn vào trace — định nghĩa các câu hỏi cụ thể bạn sẽ trả lời với trace trước khi đầu tư vào đo lường

Tri Thức và Bộ Nhớ

Cognee

Làm gì: Hệ thống bộ nhớ knowledge graph cho AI agent. Nhận tài liệu, trích xuất thực thể và quan hệ, lưu trong graph database (Neo4j), và cung cấp tìm kiếm ngữ nghĩa qua vector embedding (LanceDB).

Khi nào dùng: - Các agent cần bộ nhớ bền vững có cấu trúc về tri thức miền - Các hệ thống nơi quan hệ giữa các thực thể quan trọng (không chỉ similarity tài liệu) - Workflow agent đa phiên nơi context phải sống sót qua các lần khởi động lại

Các gotcha quan trọng: - Không có multi-tenancy tích hợp tính đến tháng 3 năm 2026 — namespace tất cả thực thể với tenant identifier thủ công (xem Chương 16) - Xây dựng đồ thị từ các tài liệu lớn chậm và tốn kém; batch ingestion trong giờ thấp điểm - Chất lượng tìm kiếm ngữ nghĩa giảm nếu embedding model dùng trong quá trình nhập khác với model dùng trong quá trình truy vấn — giữ embedding model nhất quán

Firecrawl

Làm gì: Dịch vụ web scraping được tối ưu hóa cho nhập liệu AI. Thu thập các trang web và trả về Markdown sạch phù hợp cho LLM context, xử lý JavaScript rendering, xác thực, và giới hạn tốc độ.

Khi nào dùng: - Nhập tài liệu web vào knowledge base - Giám sát website để phát hiện thay đổi - Tác vụ nghiên cứu đòi hỏi nội dung web hiện tại

Các gotcha quan trọng: - Chi phí API tích lũy nhanh cho các crawl lớn; đặt giới hạn `maxDepth` và `maxPages` rõ ràng - Các SPA nặng JavaScript có thể không render đúng mà không có cấu hình bổ sung; kiểm tra các site đích trước khi cam kết với pipeline - Tuân thủ `robots.txt` — Firecrawl tuân thủ theo mặc định; đừng vô hiệu hóa điều này

Crawl4AI

Làm gì: Giải pháp thay thế mã nguồn mở cho Firecrawl. Web crawler tự host được tối ưu cho trích xuất dữ liệu AI, với trích xuất nội dung dựa trên LLM và async crawling.

Khi nào dùng: - Khi bạn cần web crawling on-premises mà không có chi phí API bên thứ ba - Volume crawling cao nơi chi phí Firecrawl trở nên cấm đoán - Logic trích xuất tùy chỉnh mà API Firecrawl không hỗ trợ

Các gotcha quan trọng: - Yêu cầu Playwright để JavaScript rendering — thêm độ phức tạp triển khai đáng kể và memory overhead (~500MB mỗi browser instance) - Async crawling với nhiều phiên đồng thời có thể áp đảo server đích; triển khai độ trễ lịch sự - Chất lượng trích xuất thay đổi đáng kể theo cấu trúc site; kiểm tra và điều chỉnh extraction prompt theo từng domain

Neo4j

Làm gì: Graph database. Lưu trữ dữ liệu dưới dạng node và quan hệ với properties, và truy vấn nó với Cypher (ngôn ngữ truy vấn đồ thị). Được dùng bởi Cognee như backend knowledge graph.

Khi nào dùng: - Khi graph traversal là trung tâm của các truy vấn (ví dụ: “tìm tất cả service phụ thuộc vào database này”) - Knowledge graph nơi quan hệ thực thể mang ý nghĩa - Phân tích mạng/xã hội trên các mẫu tương tác agent

Các gotcha quan trọng: - Hiệu suất truy vấn Cypher giảm mạnh mà không có các index phù hợp — tạo index trên các node property được truy vấn thường xuyên tại thời điểm tạo schema - Cấu hình bộ nhớ Neo4j (`heap.initial_size`, `pagecache.size`) mặc định quá bảo thủ cho production; điều chỉnh dựa trên kích thước đồ thị - Community Edition chỉ có single-instance; clustering yêu cầu Enterprise (trả phí)

LanceDB

Làm gì: Embedded vector database được xây dựng cho ứng dụng AI. Lưu trữ vector embedding cùng với dữ liệu có cấu trúc và cung cấp tìm kiếm ANN (approximate nearest neighbor) nhanh. Được dùng bởi Cognee để tìm kiếm ngữ nghĩa.

Khi nào dùng: - Vector similarity search để truy xuất bộ nhớ ngữ nghĩa - Khi bạn muốn database nhúng (không có tiến trình server riêng) để đơn giản hóa - Ứng dụng local-first có thể cuối cùng cần cloud sync

Các gotcha quan trọng: - LanceDB được nhúng — mỗi tiến trình mở thư mục database trực tiếp; ghi đồng thời từ nhiều tiến trình yêu cầu điều phối - Rebuild index (cho IVF_PQ index) bắt buộc sau khi bổ sung dữ liệu đáng kể để duy trì chất lượng tìm kiếm; lên lịch này trong thời gian lưu lượng thấp - Tham số `metric` (cosine vs. L2 vs. dot product) phải khớp với những gì embedding model mong đợi; không khớp tạo ra kết quả tìm kiếm vô nghĩa

Bảo Mật

HashiCorp Vault

Làm gì: Nền tảng quản lý secret. Lưu trữ, xoay vòng, và kiểm toán quyền truy cập vào credentials, API key, certificate, và các secret khác. Hỗ trợ tạo secret động (ví dụ: credentials database tạm thời).

Khi nào dùng: - Bất kỳ credential mà nhiều hơn một người hoặc service cần truy cập - Credentials cần xoay vòng tự động - Môi trường yêu cầu kiểm toán nơi bạn cần biết ai đã truy cập secret nào và khi nào

Các gotcha quan trọng: - Storage backend của Vault phải có tính khả dụng cao; sự cố Vault có nghĩa là các agent không thể lấy secret và sẽ thất bại — chạy Vault ở HA mode (Raft integrated storage) - Root token được tạo ra khi khởi tạo nên được dùng một lần để cấu hình AppRole auth, rồi thu hồi và lưu trữ offline - Gia hạn lease cho dynamic secret phải được xử lý bởi client; các secret hết hạn mà không được gia hạn gây ra lỗi credential đột ngột

UFW (Uncomplicated Firewall)

Làm gì: Front-end cho iptables/nftables trên Linux. Đơn giản hóa quản lý firewall rule với cú pháp dễ đọc.

Khi nào dùng: Trên mọi server, từ ngày đầu tiên. Xem Chương 16, Sai lầm #6.

Các gotcha quan trọng: - `ufw enable` có hiệu lực ngay lập tức — đảm bảo SSH rule của bạn đã có sẵn trước khi bật, hoặc bạn sẽ tự lock out - UFW rule tồn tại qua reboot theo mặc định; các rule bạn thêm để debug tạm thời sẽ tồn tại trừ khi được xóa rõ ràng - `ufw status verbose` hiển thị rule theo định dạng dễ đọc; `ufw status numbered` hiển thị chúng với index để xóa

WireGuard VPN

Làm gì: Giao thức VPN hiện đại, hiệu suất cao được triển khai như Linux kernel module. Đơn giản hơn để cấu hình so với OpenVPN, nhanh hơn, và có attack surface nhỏ hơn.

Khi nào dùng: - Bảo mật giao tiếp giữa các agent server qua datacenter - Cung cấp kênh truy cập được mã hóa để quản trị - Thay thế các tùy chọn private networking đắt tiền từ nhà cung cấp đám mây

Các gotcha quan trọng: - WireGuard không có kết nối — không có khái niệm về peer “đã kết nối”; dùng `wg show` và ping để xác minh kết nối - Cấu hình MTU sai gây ra mất packet thậm lặng cho payload lớn; đặt MTU interface ở 1420 là mặc định an toàn - Xoay vòng key đòi hỏi cập nhật phối hợp trên cả hai peer — xây dựng quy trình xoay vòng trước khi bạn cần nó

AI API

Anthropic (Claude)

Các model: Opus 4.6, Sonnet 4.6, Haiku 4.5. Xem Phụ lục B để biết giá.

Khi nào dùng: - Suy luận phức tạp và giải quyết vấn đề đa bước (Opus) - Tác vụ agent đa năng, tạo code, phân tích (Sonnet) - Phân loại, định tuyến, định dạng, trích xuất đơn giản (Haiku)

Các gotcha quan trọng: - Giới hạn tốc độ là per API key, không per model — đột biến trên Haiku có thể ảnh hưởng đến khả năng dùng Sonnet trên cùng key - Header beta cho extended thinking (interleaved-thinking-2025-05-14) bắt buộc để truy cập reasoning trace; lên kế hoạch cho điều này trong khởi tạo client - Schema tool use phải là valid JSON Schema; schema không hợp lệ tạo ra lỗi khó hiểu thay vì thông báo validation hữu ích - Batch API (giảm 50%) có SLA xử lý 24 giờ; không phù hợp cho yêu cầu độ trễ facing user

OpenAI (GPT)

Các model: GPT-4o, GPT-4.1, o3, o4-mini. Xem Phụ lục B để biết giá.

Khi nào dùng: - Tác vụ hệ sinh thái JavaScript/TypeScript (dữ liệu training mạnh) - Tác vụ vision cùng với text (GPT-4o multimodal) - Suy luận phức tạp với extended thinking budget (o3) - Tác vụ suy luận nhạy cảm với chi phí (o4-mini)

Các gotcha quan trọng: - Reasoning token o3/o4-mini được tính phí nhưng không được trả về trong phản hồi API theo mặc định; bật `include_reasoning` để xem chúng (tăng chi phí output) - GPT-4.1 có context window 1M token không có phí tăng giá long-context — tỷ lệ giá/chiều dài context tốt nhất trong dòng GPT - Giới hạn sử dụng cấp tổ chức áp dụng trên tất cả API key trong tổ chức; các key chung có thể chạm giới hạn tổ chức bất ngờ

Google (Gemini)

Các model: Gemini 2.5 Pro, Gemini 2.5 Flash. Xem Phụ lục B để biết giá.

Khi nào dùng: - Xử lý tài liệu dài (lên đến 1M token, với 2M trong preview) - Tác vụ multimodal với video, audio, hoặc hình ảnh - Phân tích log và khớp mẫu ở quy mô (Flash nhanh và rẻ) - Tác vụ nơi chi phí là ràng buộc chính

Các gotcha quan trọng: - Gemini 2.5 Pro với `thinking_budget` bật tính phí thinking token theo tỷ lệ khác với output token; đọc tài liệu billing cẩn thận - Long-context pricing là 2x cho prompt trên 200K token — điều này không được hiển thị nổi bật; tính toán chi phí trước khi giả định long-context là kinh tế - Vertex AI và Google AI Studio dùng các API endpoint và xác thực khác nhau; chọn một cái và nhất quán

DeepSeek

Các model: R1, V3.2. Xem Phụ lục B để biết giá.

Khi nào dùng: - Khối lượng công việc nhạy cảm với chi phí nơi chất lượng chấp nhận được ở một phần nhỏ giá Claude/GPT - Tác vụ nặng về suy luận nơi cách tiếp cận chain-of-thought của R1 phù hợp tốt - Triển khai tự host (weights là open)

Các gotcha quan trọng: - Tính khả dụng API endpoint của DeepSeek từ các vùng ngoài châu Á đôi khi không ổn định; triển khai retry và fallback sang provider khác - Phiên bản tự host của R1 yêu cầu phần cứng đáng kể (tối thiểu 80GB VRAM cho full precision); các phiên bản quantized khả dụng nhưng giảm chất lượng - Định dạng output với schema tool-use phức tạp kém đáng tin cậy hơn Claude hoặc GPT; xác thực output tích cực hơn

Mistral

Các model: Mistral Nemo (và các model khác). Xem Phụ lục B để biết giá.

Khi nào dùng: - Yêu cầu data residency Châu Âu (Mistral là công ty Pháp với hạ tầng có trụ sở EU) - Tác vụ chi phí cực thấp nơi yêu cầu chất lượng khiêm tốn - Triển khai tự host với GPU hardware tiêu dùng (Mistral Nemo chạy trên 16GB VRAM)

Các gotcha quan trọng: - Context window nhỏ hơn so với các sản phẩm Claude hoặc GPT; hãy rõ ràng về context budget trong prompt - Mistral Nemo đa ngôn ngữ nhưng hoạt động tốt hơn đáng kể bằng tiếng Anh và tiếng Pháp so với các ngôn ngữ khác

Proxy và Cân Bằng Tải

API Proxy Service

Làm gì: Định tuyến AI API request qua nhiều provider với logic fallback, xử lý giới hạn tốc độ, và theo dõi chi phí. Cung cấp API surface thống nhất để code agent không cần các client library dành riêng cho provider.

Khi nào dùng: - Thiết lập đa provider nơi bạn muốn fallback tự động khi một provider bị rate-limit - Theo dõi chi phí tập trung trên các provider - A/B testing các model khác nhau trên cùng loại tác vụ

Các gotcha quan trọng: - Thay đổi cấu hình đòi hỏi kiểm tra cẩn thận — như đã lưu ý trong Chương 16, một agent sửa đổi cấu hình API proxy đã gây ra sự cố đáng kể. Coi cấu hình API proxy là infrastructure, không phải application config - Thứ tự fallback quan trọng: nếu fallback của bạn là model đắt hơn, sự cố provider chính sẽ thâm lạng tăng chi phí - Endpoint kiểm tra sức khỏe cho mỗi provider backend cần được cấu hình rõ ràng; nếu không, một provider bị lỗi tiếp tục nhận lưu lượng cho đến khi các request thất bại

API Provider Manager

Làm gì: Lốp lập lịch và cân bằng tải request cho các lệnh gọi AI API. Quản lý queue request, priority lane, và thực thi ngân sách qua agent mesh.

Khi nào dùng: - Điều phối sử dụng API qua nhiều agent chia sẻ giới hạn tốc độ - Thực thi giới hạn ngân sách per-agent hoặc per-task ở cấp hạ tầng (thay vì dựa vào application code) - Lập lịch request theo ưu tiên (request facing user được ưu tiên cao hơn background job)

Các gotcha quan trọng: - Priority queue có thể làm đổi request ưu tiên thấp trong thời gian tải cao điểm; đảm bảo background job có throughput được đảm bảo tối thiểu - Tích hợp yêu cầu các agent định tuyến tất cả API call qua manager, không trực tiếp đến provider — thay đổi kiến trúc khó để retrofit

Giao Tiếp

Telegram Bot

Làm gì: Gửi thông báo, alert, và các lệnh tương tác đến các kênh hoặc nhóm Telegram qua Telegram Bot API.

Khi nào dùng: - Cập nhật trạng thái agent và alert thời gian thực - Kiểm soát agent tương tác (gửi lệnh qua tin nhắn Telegram) - Thông báo chi phí và ngân sách - Alert sự cố với các nút hành động

Các gotcha quan trọng: - Bot API bị giới hạn tốc độ 30 tin nhắn/giây đến một chat đơn lẻ và 1 tin nhắn/giây đến cùng một nhóm — các alert storm sẽ bị throttle thâm lạng - Định dạng Markdown trong tin nhắn Telegram dùng hương vị không chuẩn (MarkdownV2); kiểm tra chuỗi định dạng cẩn thận hoặc dùng `parse_mode=None` và plain text - Bot token trong biến môi trường vẫn là credentials; xoay vòng chúng nếu bị lộ, lưu trong Vault

```

# Gửi alert Telegram tối giản
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]} # Giới hạn 4096 ký tự Telegram
    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

```

Nhắn Tin Liên Agent Dựa Trên tmux

Làm gì: Cơ chế IPC nhẹ dùng các phiên tmux được đặt tên như các endpoint tin nhắn. Một agent ghi vào file chung; phiên của agent đích poll file hoặc nhận `tmux send-keys injection`.

Khi nào dùng: - Giao tiếp liên agent cục bộ trên cùng server - Phát triển và prototyping điều phối agent trước khi đầu tư vào message queue - Kênh giao tiếp dự phòng khi HTTP-based IPC không khả dụng

Các gotcha quan trọng: - Các phiên tmux được gắn với một người dùng cụ thể; nhắn tin liên agent qua người dùng đòi hỏi đường dẫn hệ thống file chung với quyền phù hợp - `tmux send-keys` tiêm tổ hợp phím vào pane đang hoạt động — nếu phiên agent có một lệnh khác được focus, các phím đến sai tiến trình - Cách tiếp cận này không mở rộng vượt ra ngoài một server duy nhất; nếu bạn cần cross-server IPC, dùng Redis Streams hoặc tương đương

Để biết tài liệu hiện tại về bất kỳ công cụ nào được liệt kê ở đây, dùng kỹ năng `docs-seeker` hoặc tài liệu chính thức của dự án. Phiên bản và tính năng thay đổi; các gotcha thường ở lại như vậy.

Phụ Lục B: Tài Liệu Tham Khảo Giá AI Model (Tháng 3 Năm 2026)

Giá thay đổi. Kiểm tra trang giá của nhà cung cấp trước khi cam kết với ngân sách. Phụ lục này phản ánh giá công khai tính đến ngày 1 tháng 3 năm 2026.

Tất cả giá tính bằng USD trên **triệu token** (MTok) trừ khi có ghi chú. Input = token gửi đến model (prompt + context). Output = token được model trả về (completion).

Tham Khảo Nhanh: Giá Mỗi MTok

Model	Input	Output	Context	Ghi chú
Claude Opus 4.6	\$5.00	\$25.00	200K (1M beta)	Suy luận tốt nhất
Claude Sonnet 4.6	\$3.00	\$15.00	200K	Đa năng
Claude Haiku 4.5	\$1.00	\$5.00	200K	Nhanh, rẻ
GPT-4o	\$2.50	\$10.00	128K	Multimodal mạnh
GPT-4.1	\$2.00	\$8.00	1M	Giá/context tốt nhất
o3	\$10.00	\$40.00	200K	Extended thinking
o4-mini	\$1.10	\$4.40	200K	Suy luận rẻ
Gemini 2.5 Pro	\$1.25	\$10.00	1M (2M preview)	Context dài
Gemini 2.5 Flash	\$0.30	\$2.50	1M	Nhanh nhất/rẻ nhất
DeepSeek R1	\$0.55	\$2.19	128K	Open weights
DeepSeek V3.2	\$0.14	\$0.28	128K	Cực rẻ
Llama 4	\$0.27	\$0.85	128K	Open source
Mistral Nemo	\$0.02	\$0.02	128K	Tùy chọn EU-hosted

Claude (Anthropic)

Bảng Giá

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

Giảm Giá và Hệ Số Nhân

Batch API (giảm 50%): Tất cả model Claude hỗ trợ Batch API để xử lý bất đồng bộ với SLA 24 giờ.

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 (hệ số nhân 2x trên 200K token): Khi dùng 1M token extended context beta trên Opus 4.6, các token vượt quá 200K standard window được tính phí gấp 2x giá cơ bản.

Token trong prompt	Tỷ lệ input Opus 4.6
0 - 200K	\$5.00/MTok
200K - 1M	\$10.00/MTok

Prompt caching: Anthropic hỗ trợ prompt caching cho các system prompt lặp lại và context prefix lớn. - Cache write: 1.25x tỷ lệ input cơ bản - Cache read: 0.1x tỷ lệ input cơ bản (tiết kiệm 90% trên token được cache)

```
# Tính chi phí với 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

```

Hướng Dẫn Lựa Chọn Model Claude

- Đây có phải yêu cầu facing user đòi hỏi phản hồi nhanh không?
- └ Có → Haiku 4.5 (nếu đơn giản) hoặc Sonnet 4.6 (nếu phức tạp)
 - └ Không → Bạn có thể batch nó không?
 - └ Có → Batch API (tiết kiệm 50%)
 - └ Không → Tiếp tục kiểm tra độ phức tạp
- Nó có đòi hỏi suy luận đa bước sâu hoặc giải quyết vấn đề mới không?
- └ Có → Opus 4.6
 - └ Không → Sonnet 4.6 cho hầu hết tác vụ, Haiku 4.5 cho phân loại/định tuyến

OpenAI (GPT)

Bảng Giá

Model	Input (/MTok) Output(/MTok)	Context	Ghi chú	
GPT-4o	\$2.50	\$10.00	128K	Multimodal (vision, audio)
GPT-4.1	\$2.00	\$8.00	1M	Không có phí tăng long-context
o3	\$10.00	\$40.00	200K	Model extended thinking
o4-mini	\$1.10	\$4.40	200K	Suy luận tiết kiệm chi phí

Giảm Giá và Hệ Số Nhân

Cached input (tiết kiệm 50% khi cache hit):

Model	Input Chuẩn	Input Cached
GPT-4o	\$2.50	\$1.25
GPT-4.1	\$2.00	\$1.00

Model	Input Chuẩn	Input Cached
o3	\$10.00	\$5.00
o4-mini	\$1.10	\$0.55

Reasoning token o3/o4-mini: Reasoning token được tạo ra nội bộ trong quá trình extended thinking và được tính phí theo tỷ lệ output token. Chúng không được trả về trong phản hồi theo mặc định. Bật `include_reasoning: true` trong API request để nhận chúng (lưu ý: điều này tăng output token được tính phí).

Lợi thế context GPT-4.1: GPT-4.1 là model context 1M duy nhất trong dòng GPT không có phí tăng giá long-context. Với các khối lượng công việc có prompt từ 200K đến 1M token, GPT-4.1 thường là tùy chọn tiết kiệm chi phí nhất trên tất cả các nhà cung cấp lớn.

Batch API OpenAI

Batch API của OpenAI cung cấp giảm 50% với cửa sổ hoàn thành 24 giờ, khớp với sản phẩm của Anthropic.

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)

Bảng Giá

Model	Input $\leq 200K$ (/ 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

Giảm Giá và Hệ Số Nhân

Long-context (hệ số nhân 2x trên 200K token): Không giống GPT-4.1, Gemini áp dụng hệ số nhân 2x cho input token vượt quá 200K. Với context window rất lớn, hãy tính toán cẩn thận.

```
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 (giảm 50%): API xử lý batch của Gemini cung cấp giảm 50% trên tất cả chi phí token với xử lý bất đồng bộ.

Model	Batch Input $\leq 200K$	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 hỗ trợ thinking budget có thể cấu hình (0 đến 32K thinking token). Thinking token được tính phí theo tỷ lệ output (\$10.00/MTok). Vô hiệu hóa thinking cho các tác vụ đơn giản hơn để giảm chi phí đáng kể.

Free Tier (Google AI Studio)

Gemini 2.5 Flash cung cấp free tier qua Google AI Studio: - 1.500 yêu cầu/ngày miễn phí - Giới hạn tốc độ 1M token/phút (tăng trả phí) - Không phù hợp cho production; dùng Vertex AI cho các triển khai production

Open Source / Tự Host

API Pricing (qua API provider)

Model	Input (/MTok) Output(/MTok)	Context	Ghi chú	
DeepSeek R1	\$0.55	\$2.19	128K	Model suy luận, open weights
DeepSeek V3.2	\$0.14	\$0.28	128K	Model open tốt nhất về giá/hiệu suất
Llama 4 (Scout)	\$0.27	\$0.85	128K	Meta open source
Mistral Nemo	\$0.02	\$0.02	128K	EU-hosted, cực rẻ

Chi Phí Tự Host

Khi tự host các model open, chi phí API được thay thế bằng chi phí phần cứng:

DeepSeek R1 (Full, 671B tham số):

Phần cứng cần thiết: 8× H100 80GB (hoặc tương đương)
Chi phí đám mây (ví dụ Lambda Labs): ~\$25/giờ cho 8× H100
Throughput: ~15-20 token/giây ở full precision
Điểm hòa vốn so với API: $\sim \$25/\text{giờ} \div (15 \text{ tok/s} \times 3600 \text{ s/giờ}) = \sim \$0.46/1\text{K token tạo ra}$
Tự host tiết kiệm chi phí trên ~10M output token/ngày

DeepSeek R1 Distill (8B, dựa trên Llama):

Phần cứng cần thiết: 1× RTX 4090 (24GB VRAM)
Chi phí đám mây: ~\$0.80/giờ
Throughput: ~60-80 token/giây
Khả năng giảm so với R1 đầy đủ, nhưng phù hợp với nhiều tác vụ

Mistral Nemo (12B):

Phần cứng cần thiết: 1x RTX 4090 hoặc 2x RTX 3090
Chạy trên GPU tiêu dùng – phù hợp cho triển khai on-premises
Trần chất lượng rất thấp nhưng đủ cho phân loại, định tuyến, định dạng

Ước Tính Chi Phí Hàng Tháng Theo Tải Agent

Các ước tính sau dùng phân phối token thực tế: - Token input trung bình mỗi yêu cầu: 2.500 - Token output trung bình mỗi yêu cầu: 600 - Tháng 30 ngày

1M Token/Ngày (~400 yêu cầu/ngày)

Provider + Model	Chi phí ngày	Chi phí tháng	Ghi chú
Claude Haiku 4.5	\$1.76	\$52.80	Tỷ lệ 70% input, 30% output
Claude Sonnet 4.6	\$5.25	\$157.50	Khối lượng công việc chung
Claude Opus 4.6	\$8.75	\$262.50	Chất lượng đầy đủ
GPT-4o	\$4.43	\$132.75	
GPT-4.1	\$3.55	\$106.50	
Gemini 2.5 Flash	\$0.53	\$15.90	Giá trị tốt nhất ở quy mô này
DeepSeek V3.2	\$0.25	\$7.50	Cực rẻ

Khuyến nghị ở 1M token/ngày: Kết hợp Gemini 2.5 Flash (tác vụ hàng loạt) + Claude Sonnet 4.6 (tác vụ chất lượng). Ước tính tháng kết hợp: ~\$60-90.

10M Token/Ngày (~4.000 yêu cầu/ngày)

Provider + Model	Chi phí ngày	Chi phí tháng	Ghi chú
Claude Haiku 4.5	\$17.60	\$528	
Claude Sonnet 4.6	\$52.50	\$1.575	
Claude Opus 4.6	\$87.50	\$2.625	Đắt ở quy mô
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	

Khuyến nghị ở 10M token/ngày: Định cỡ model đúng trở nên quan trọng. Chi phí kết hợp production điển hình với định tuyến tích cực (60% Haiku/Flash, 35% Sonnet, 5% Opus/o3): ~\$350-500/tháng.

100M Token/Ngày (~40.000 yêu cầu/ngày)

Ở quy mô này, định cỡ model và batch API không phải tùy chọn — chúng là yêu cầu sống còn.

Provider + Model	Chi phí ngày	Chi phí tháng	Ghi chú
Claude Sonnet 4.6	\$525	\$15.750	Toàn bộ Sonnet, không tối ưu
Claude Sonnet 4.6 (Batch)	\$262.50	\$7.875	Giảm 50% batch
Gemini 2.5 Flash	\$53	\$1.590	Toàn bộ Flash
Gemini 2.5 Flash (Batch)	\$26.50	\$795	
DeepSeek V3.2	\$25.20	\$756	Chi phí API thấp nhất
Tự host R1 Distill	~\$580	~\$17.400	Phần cứng, không phải token

Khuyến nghị ở 100M token/ngày:

Chiến lược định tuyến tác vụ cho 100M token/ngày:

- 60% → Gemini 2.5 Flash (batch nơi có thể)
\$0.15/MTok batch → ~\$2.700/tháng
- 30% → Claude Sonnet 4.6 (batch nơi có thể)
\$1.50/MTok batch → ~\$6.750/tháng
- 8% → DeepSeek V3.2 (hàng loạt nhảy cảm chi phí)
\$0.14/MTok → ~\$336/tháng
- 2% → Claude Opus 4.6 hoặc o3 (suy luận phức tạp)
\$5.00/MTok → ~\$300/tháng

Tổng ước tính: ~\$10.086/tháng

so với toàn bộ Sonnet không tối ưu: \$47.250/tháng

Tiết kiệm: 79%

Kỹ Thuật Tối Ưu Chi Phí

1. Prompt Caching

Với các agent có system prompt dài hoặc context lặp lại (tài liệu infrastructure, tóm tắt codebase), prompt caching là tối ưu hóa có đòn bẩy cao nhất:

```
# Anthropic cache control headers
response = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=1024,
    system=[
        {
            "type": "text",
            "text": STATIC_SYSTEM_PROMPT, # Hiếm khi thay đổi
            "cache_control": {"type": "ephemeral"} # Cache trong 5 phút
        }
    ],
)
```

```

        messages=[{"role": "user", "content": user_message}]
    )

    # Kiểm tra hiệu suất cache
    cache_creation_tokens = response.usage.cache_creation_input_tokens
    cache_read_tokens = response.usage.cache_read_input_tokens
    # cache_read ở tỷ lệ 0.1x so với cache_creation ở tỷ lệ 1.25x

```

Tiết kiệm điển hình: 40-70% chi phí input token cho các agent có system prompt lớn, ổn định.

2. Kiểm Soát Output Token

Output token tốn gấp 3-8x hơn mỗi token so với input token trên tất cả nhà cung cấp. Hạn chế độ dài output trực tiếp giảm chi phí.

```

# Rõ ràng về định dạng và độ dài output
CONCISE_SUFFIX = ""
Chỉ trả lời bằng JSON. Không giải thích. Tối đa 200 token.
Schema: {"status": "ok|error", "action": "string", "details": "string"}
"""

# Xấu: "Phân tích cấu hình này và cho tôi biết bạn nghĩ gì"
# Tốt: f"Phân loại mức độ nghiêm trọng vấn đề cấu hình này (critical/high/medium/low).
{CONCISE_SUFFIX}"

```

3. Quản Lý Context Window

Context tăng không giới hạn là nguyên nhân phổ biến nhất của các đột biến chi phí bất ngờ trong các phiên agent chạy dài.

```

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:
            # Giữ system message + N tin nhắn cuối
            self._messages = self._messages[:1] + self._messages[-self.keep_recent:]

    @property
    def messages(self) -> list[dict]:
        return self._messages

```

4. Schema Output Có Cấu Trúc

Yêu cầu output có cấu trúc (JSON mode, tool use schema) giảm độ dài output và loại bỏ các token dùng cho đóng khung hội thoại:

```
Không có schema: "Dựa trên phân tích của tôi về cấu hình được cung cấp, tôi thấy rằng
                  có một số vấn đề cần được giải quyết. Đầu tiên,
                  cài đặt max_connections..." (150 token)

Với schema:      {"issues": ["max_connections quá thấp", "shared_buffers không đủ"],
                  "severity": "high"} (25 token)
```

Tiết kiệm: 60-85% output token cho các tác vụ dữ liệu có cấu trúc.

Theo Dõi Giá: Những Gì Thay Đổi và Những Gì Cần Giám Sát

Giá trong không gian này thay đổi thường xuyên hơn các cloud service truyền thống. Đặt nhắc nhở lịch để đánh giá lại giá hàng quý. Những điều thường thay đổi:

- **Tỷ lệ input/output** — các nhà cung cấp định kỳ điều chỉnh tỷ lệ giữa input và output pricing khi họ tối ưu hóa hạ tầng serving
- **Phát hành model mới** — model Haiku hoặc Flash-tier mới thường cung cấp khả năng tốt hơn với giá tương tự hoặc thấp hơn; đánh giá lại các quyết định định tuyến sau mỗi lần phát hành
- **Mở rộng Batch API** — không phải tất cả model đều hỗ trợ batch API khi ra mắt; kiểm tra các bổ sung mới
- **Sửa đổi free tier** — Google AI Studio free tier thay đổi; đừng xây dựng phụ thuộc production trên free tier
- **Long-context pricing** — khi chi phí serving giảm, phí tăng long-context (hiện tại 2x cho Gemini, Claude) có thể giảm hoặc bị loại bỏ

```
# Script kiểm tra chi phí nhanh – chạy hàng tháng
#!/bin/bash
# cost-check.sh: Lấy 30 ngày sử dụng token từ mỗi 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}'
```

Dữ liệu giá lấy từ tài liệu chính thức của nhà cung cấp. Luôn xác minh giá hiện tại tại trang giá của nhà cung cấp trước khi cam kết ngân sách. Giảm giá Batch API và caching áp dụng trên giá cơ bản được liệt kê.

Phụ lục C: Checklist Môi Trường Production

In chúng ra. Ép plastic nếu bạn thích sự kịch tính. Điều quan trọng là phải chạy qua từng mục — không phải chỉ đọc một lần rồi tin vào trí nhớ.

Mỗi checklist được thiết kế để dùng trực tiếp. Các mục đánh dấu **[BLOCKER]** phải hoàn thành trước khi tiếp tục. Các mục đánh dấu **[RECOMMENDED]** được khuyến nghị mạnh nhưng tùy thuộc vào ngữ cảnh.

1. Checklist Trước Khi Deploy

Chạy checklist này trước mỗi lần deploy agent mới hoặc thay đổi agent đáng kể lên production.

Code và Configuration

[BLOCKER] Tất cả secrets đều nằm trong Vault hoặc biến môi trường — không có credentials hardcoded trong code hoặc config file

[BLOCKER] `git log --all --grep="sk-ant\|api_key\|password\|secret"` không trả về kết quả nào

[BLOCKER] System prompt của agent đã được review để phát hiện các quyền hạn không mong muốn (quyền ghi, thay đổi infra, truy cập credentials)

Model được chọn phù hợp với độ phức tạp của task — không mặc định dùng Opus cho mọi thứ

Output token limit đã được đặt cho tất cả các API call (tham số `max_tokens` rõ ràng)

Quản lý context window đã được triển khai — không có message history không giới hạn

Tool use schemas đã được validate theo JSON Schema spec

Error handling bao gồm: API rate limits, timeouts, malformed responses, budget exceeded

Testing

[BLOCKER] Unit tests pass cho tất cả các tool implementation của agent

[BLOCKER] Integration test với real API call (không phải mock) cho workflow chính của agent

Các edge case đã được test: input rỗng, input rất dài, adversarial input

Validation output của sub-agent đã được test với các response cố tình bị lỗi format

Budget guard đã được test — xác nhận nó dừng execution khi đạt giới hạn

Retry logic đã được test — xác nhận nó không loop vô hạn khi gặp lỗi liên tục

Infrastructure

[BLOCKER] UFW rules đã được cập nhật để cho phép các port mới của agent (nếu có)

[BLOCKER] Database migrations đã được apply lên staging và verify

Docker image đã được build, tag với git commit SHA, và push lên registry

Resource limits đã được đặt trong container config (`--memory` , `--cpus` , `--pids-limit`)

Health check endpoint đã được triển khai và phản hồi tại `/health`

Cấu hình routing của Traefik/HAProxy đã được cập nhật và test trên staging

Tính nhất quán của config etcd đã được verify trên tất cả các cluster node (nếu có Patroni cluster)

Observability

[BLOCKER] Prometheus metrics endpoint đã được expose và scraping thành công

[BLOCKER] Daily budget alert đã được cấu hình (notification ngưỡng 80%)

[BLOCKER] Hard budget ceiling đã được cấu hình (ngưỡng 100% — dừng hoặc alert)

Grafana dashboard đã được cập nhật cho agent mới hoặc metrics mới

Alertmanager routing đã được verify — test alert kích hoạt đến đúng channel

Log format là structured JSON (machine-parseable bởi Loki)

Agent ID được bao gồm trong tất cả các dòng log và metric label

Thông Báo

Progress reporting đã được triển khai cho các task > 30 giây

Thông báo Telegram (hoặc channel ưu tiên) đã được test end-to-end

Error notification bao gồm: agent ID, task ID, error message, timestamp

Thông báo deploy đã được gửi đến team channel

Kế Hoạch Rollback

Docker image tag trước đó đã được ghi lại: _____

Rollback command đã được document và test trên staging

Quy trình database rollback đã được document (nếu có schema changes)

Người quyết định rollback đã được xác định: _____

Thời gian giới hạn để quyết định rollback: _____ phút sau deploy

2. Checklist Bảo Mật Hệ Thống

Chạy một lần khi thiết lập server ban đầu, và review lại hàng quý.

Network

[BLOCKER] UFW đã bật: `sudo ufw status` hiển thị "Status: active"

[BLOCKER] Default incoming policy là DENY: `sudo ufw default deny incoming`

[BLOCKER] SSH access chỉ giới hạn ở các IP management range đã biết

[BLOCKER] PostgreSQL port (5432) chỉ có thể truy cập từ IP của application server

[BLOCKER] etcd ports (2379, 2380) chỉ có thể truy cập trong phạm vi IP của cluster node

Tất cả các port không dùng đã được đóng — chạy `sudo ss -tlnp` và verify từng port đang mở

WireGuard VPN đã được cấu hình cho giao tiếp agent xuyên datacenter

Port public-facing: chỉ 80 (redirect) và 443 (TLS) cho web endpoint

Patroni REST API (8008) không được expose ra internet công cộng

Authentication và Secrets

[BLOCKER] Vault đã được khởi tạo và unsealed; root token đã bị thu hồi sau khi thiết lập ban đầu

[BLOCKER] AppRole auth đã được cấu hình cho từng agent service

Tất cả API key được lưu trong Vault; các ứng dụng đọc qua Vault agent sidecar

Database password được tạo ngẫu nhiên (tối thiểu 32 ký tự), lưu trong Vault

SSH keys: password authentication đã bị vô hiệu hóa (PasswordAuthentication no trong sshd_config)

SSH root login đã bị vô hiệu hóa (PermitRootLogin no)

SSH keys đã được rotate trong vòng 12 tháng qua

Telegram bot token được lưu trong Vault (không lưu trong file .env)

File .env có trong .gitignore — verify bằng `git check-ignore .env`

System

OS package đã được cập nhật: `sudo apt update && sudo apt upgrade -y`

Automatic security update đã bật: `sudo dpkg -l unattended-upgrades`

Fail2ban đã được cài đặt và cấu hình để bảo vệ khỏi brute-force SSH

Các agent process chạy dưới user low-privilege chuyên dụng (không phải root, không phải user của bạn)

Docker daemon không thể truy cập bởi agent user (dùng rootless Docker hoặc strict socket permissions)

/tmp được mount với option noexec (ngăn chặn thực thi script từ temp file)

Audit logging đã bật: `sudo systemctl status auditd`

TLS và Certificates

Tất cả HTTPS endpoint dùng TLS 1.2+ (mặc định của Traefik: compliant)

Let's Encrypt certificates tự động gia hạn: kiểm tra Traefik logs để xem hoạt động gia hạn

Các internal service dùng self-signed cert đã có cert pinning hoặc CA validation

Alert giám sát certificate hết hạn đã được cấu hình (alert 30 ngày trước khi hết hạn)

Đặc Thù Agent

Agent không thể tự sửa đổi system prompt hoặc tool list khi runtime

Các đề xuất thay đổi infrastructure cần được con người phê duyệt (không ghi trực tiếp vào infra config)

Output của agent được log trước khi thực thi (audit trail)

Agent không thể truy cập các secret nằm ngoài Vault policy được giao

Tin nhắn inter-agent được validate — agent không thực thi mù quáng lệnh từ agent khác

3. Checklist Quản Lý Chi Phí

Chạy khi deploy ban đầu và hàng tháng sau đó.

Kiểm Soát Budget

[BLOCKER] Daily budget ceiling đã được cấu hình trong `CostTracker` hoặc tương đương

[BLOCKER] Alert ở 80% daily budget — kích hoạt đến Telegram hoặc Slack

[BLOCKER] Hard stop ở 100% daily budget — agent dừng nhận request mới

Monthly budget ceiling đã được đặt ở cấp cloud provider (AWS Budgets, GCP Billing alerts)

Phân bổ budget cho từng agent đã được document: _____

Budget review đã được lên lịch: thứ Hai đầu tiên của mỗi tháng

Chọn Model Phù Hợp

Bảng routing model đã được review và cập nhật

Tỷ lệ phần trăm request được route đến mỗi tier đã được document:

- Haiku/Flash tier: _____%
- Sonnet/GPT-4o tier: _____%
- Opus/o3 tier: _____%

Chi phí trung bình cho mỗi task thành công đã được tính toán: \$ _____

Các task được route đến Opus mà có thể dùng Sonnet — đã review và reroute nếu phù hợp

Hiệu Quả Token

Prompt caching đã bật cho static system prompt (Anthropic)

Cache hit rate đang được monitor: mục tiêu > 60% cho agent có system prompt ổn định

Context window trimming đã được triển khai — không có session nào vượt _____ token

Output token cap đã được đặt cho tất cả API call

Structured output schema đang được sử dụng cho task trích xuất dữ liệu

Các prompt dài dòng/khám phá đã được thay bằng prompt có mục tiêu, cụ thể

Kiểm Tra Scheduled Job

Liệt kê tất cả cron job / scheduled agent task đang hoạt động: `crontab -l` && `systemctl list-timers`

Với mỗi scheduled job, document:

- Tần suất: _____
- Ước tính chi phí token hàng ngày: \$ _____
- Lần cuối output được đọc và có hành động: _____
- Quyết định: giữ / giảm tần suất / xóa

Các job mà output không được đọc thường xuyên: **đã xóa hoặc vô hiệu hóa**

Tổng chi tiêu hàng ngày cho scheduled job: \$ _____ (mục tiêu: < 20% tổng chi tiêu hàng ngày)

Phát Hiện Cost Anomaly

Chi phí hàng ngày được review mỗi buổi sáng (tự động hóa bằng báo cáo Telegram buổi sáng)

Ngưỡng cost spike đã được định nghĩa: alert nếu chi tiêu ngày > _____ % của trung bình 7 ngày

Top 3 nguyên nhân tổn chi phí đã được xác định và document tháng này:

1. _____

2. _____

3. _____

4. Checklist Thiết Lập Monitoring

Chạy khi thiết lập server mới hoặc mở rộng monitoring stack.

Prometheus

Prometheus đang scrape tất cả agent service (verify trong Status > Targets)

Scrape interval được đặt phù hợp: 15s cho active agent, 60s cho batch job

Retention period đã được cấu hình: tối thiểu 30 ngày (--storage.tsdb.retention.time=30d)

Core agent metrics đã được triển khai và hiển thị trong Prometheus:

- agent_tokens_total (counter, labeled theo model và type)
- agent_tasks_total (counter, labeled theo status)
- agent_task_duration_seconds (histogram)
- agent_cost_usd_total (counter, labeled theo agent_id)
- agent_queue_depth (gauge)

Infrastructure metrics đang được scrape:

- Node Exporter trên tất cả server
- PostgreSQL Exporter cho database metrics

- Patroni metrics endpoint (/metrics trên port 8008)
- etcd metrics endpoint

Alertmanager

Alertmanager đã kết nối với Prometheus

Notification channel đã được cấu hình và test (Telegram/Slack/PagerDuty)

Core alert rules đã được định nghĩa:

Agent down (không có heartbeat trong > 5 phút)

Daily budget > 80% đã dùng

Error rate > 5% trong 15 phút

Queue depth > 50 trong > 10 phút

Patroni primary không khả dụng

Replication lag > 30 giây

Disk usage > 80% trên bất kỳ volume nào

etcd cluster có < 2 member healthy

Alert routing đã được verify — test kích hoạt từng alert và xác nhận nó đến đúng channel

Inhibition rules đã được cấu hình — alert high-severity suppress các alert low-severity liên quan

On-call rotation hoặc escalation path đã được document

Grafana

Dashboard đã được tạo cho tất cả agent service

Dashboard JSON đã được export và commit vào git: `dashboards/`

Core panel cho mỗi agent dashboard:

- Token consumption rate (rate trong 1h)
- Chi phí mỗi ngày (sum trong 24h window)
- Task success/error rate
- P50/P95/P99 task duration
- Queue depth theo thời gian

Infrastructure dashboard bao gồm:

- CPU/RAM/Disk cho mỗi server
- Network I/O
- PostgreSQL connections, replication lag, transaction rate
- etcd leader changes (nên bằng không khi hoạt động bình thường)

Default `admin` password đã được thay đổi

Logging (Loki)

Loki đang nhận log từ tất cả agent service

Log label có low-cardinality (service, level, agent_id — không dùng request_id hoặc user_id)

Log format là structured JSON: `{"level":"error","msg":"...","agent_id":"...","ts":"..."}`

Log retention policy đã được cấu hình (mặc định 30 ngày)

Log-based alert rule cho các log ERROR/CRITICAL có tần suất cao

5. Checklist Xử Lý Sự Cố

Dùng khi có sự cố và thời gian eo hẹp.

5 Phút Đầu: Triage

Xác định phạm vi: Đây là một agent, tất cả agent, hay infrastructure?

Kiểm tra Grafana: Metrics nào tăng đột biến hoặc giảm mạnh vào thời điểm xảy ra sự cố?

Kiểm tra Alertmanager: Những alert nào đang kích hoạt? Mức độ nghiêm trọng là gì?

Kiểm tra agent heartbeat: `systemctl status agent-* | grep Active`

Kiểm tra database: `patronictl -c /etc/patroni/patroni.yml list` — có đúng một primary không?

Thông báo: Post thông tin bắt đầu sự cố lên team channel kèm đánh giá phạm vi ban đầu

Lệnh Chẩn Đoán

```
# Agent process có đang chạy không?
systemctl status claude-code-agent
journalctl -u claude-code-agent --since "10 minutes ago" -f

# Database có thể kết nối không?
psql -h localhost -U postgres -c "SELECT version();"
patronictl -c /etc/patroni/patroni.yml list

# etcd có healthy không?
etcdctl --endpoints=http://server1:2379,http://server2:2379,http://server3:2379 \
  endpoint health

# Cái gì đang dùng nhiều memory?
free -h
ps aux --sort=-%mem | head -20

# API error rate hiện tại là bao nhiêu?
# (Prometheus query)
rate(agent_tasks_total{status="error"}[5m]) /
rate(agent_tasks_total[5m])

# Hôm nay đã chi bao nhiêu?
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}')" 
```

Kiểm Chế Sự Cố

Nếu cost runaway: vô hiệu hóa scheduled job trước (`systemctl stop agent-scheduler`)

Nếu agent bị loop: `systemctl stop <agent-service>`; không restart cho đến khi biết root cause

Nếu database failover storm: dừng Patroni trên tất cả node trước khi restart etcd

Nếu sự cố bảo mật: cô lập server bị ảnh hưởng khỏi mạng trước khi điều tra: `sudo ufw deny incoming`

Nếu sự cố từ phía API provider: kiểm tra trang status của provider trước khi debug nội bộ

Giải Quyết

Root cause đã được document bằng văn bản (dù chỉ một câu)

Fix đã được apply và test trên staging nếu có thời gian; apply trực tiếp lên production nếu critical

Agent đã được restart: `systemctl start <agent-service>`

Verify recovery: kiểm tra Grafana dashboard để xem normalization trong 10 phút

Budget check: xác nhận chi tiêu không tăng đột biến ngoài mức chấp nhận được trong sự cố

Thông báo giải quyết đến team channel kèm thời gian và tóm tắt root cause

Sau Sự Cố (trong vòng 48 giờ)

Postmortem bằng văn bản: điều gì xảy ra, timeline, root cause, fix, phòng ngừa

Gap trong monitoring đã được xác định (nếu sự cố không bị phát hiện bởi alert hiện tại, thêm alert)

Runbook đã được cập nhật với các lệnh chẩn đoán mới phát hiện trong sự cố

Checklist đã được cập nhật nếu thiếu bước nào

6. Checklist Debug Agent

Dùng khi agent tạo ra output sai, không nhất quán, hoặc bất ngờ.

Xác Nhận Vấn Đề

Bạn có thể tái hiện vấn đề một cách nhất quán không? Nếu không, có thể là model non-determinism — đặt `temperature: 0` và thử lại

Vấn đề nằm ở reasoning của agent, tool call của nó, hay output cuối cùng?

Vấn đề là mới (regression) hay nó luôn hoạt động như vậy?

Vấn đề xảy ra với tất cả input hay chỉ với input cụ thể?

Kiểm Tra Agent

```
# Bật verbose logging cho debug session
import anthropic
import json

client = anthropic.Anthropic()

# Log 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"])

```

Các Pattern Lỗi Phổ Biến

Triệu Chứng	Nguyên Nhân Có Thể	Fix
Agent từ chối task mà trước đây đã làm được	System prompt conflict hoặc safety update mới	Review system prompt để tìm instruction mơ hồ
Tool được gọi với argument sai	Schema mismatch hoặc mô tả tool không rõ ràng	Cải thiện mô tả tool; thêm ví dụ vào schema
Agent loop mà không hoàn thành	Không có điều kiện kết thúc rõ ràng	Thêm tiêu chí success/failure rõ ràng vào system prompt
Output không nhất quán với cùng input	<code>temperature > 0</code>	Đặt <code>temperature: 0</code> cho task deterministic
Agent bỏ qua kết quả tool	Tool result format không như mong đợi	Log raw tool result; verify format khớp với agent mong đợi
Context window vượt giới hạn	Message history không giới hạn	Triển khai <code>ContextManager</code> với explicit trim policy
Tỷ lệ cao "I cannot help with that"	System prompt quá restrictive	Review và nói lỏng các restriction không cần thiết
Sub-agent trả về output hợp lý nhưng sai	Không có output validation	Thêm validator cho output type này

Debug Prompt

```
# Minimal reproducible example – bỏ đi mọi thứ không cần thiết
debug_response = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=512,
    # Bắt đầu KHÔNG có system prompt – thêm lại để xác định xem nó có gây ra vấn đề không
    messages=[
        {"role": "user", "content": PROBLEMATIC_INPUT}
    ]
)
# Nếu hoạt động không có system prompt, vấn đề nằm trong system prompt
# Thêm lại system prompt, rồi tools, rồi context – binary search tìm vấn đề
```

Debug Tool Use

Tool schema là valid JSON Schema (validate tại jsonschema.org)

Mô tả tool giải thích khi nào KHÔNG nên dùng tool (overuse là lỗi thường gặp)

Tool return được format theo cách agent mong đợi (string vs object)

Tool error trả về error message hữu ích, không phải Python traceback

Tool execution được log với input và output cho mỗi lần gọi

7. Cây Quyết Định Chọn Model

Dùng khi chọn model nào để gọi cho loại task mới.

```
BẮT ĐẦU: Task chính là gì?
|
|─ Phân loại / Routing / Tagging
|   └─ Claude Haiku 4.5 hoặc Gemini 2.5 Flash
|       Độ phức tạp thấp, khối lượng cao → Flash (rẻ hơn)
|       Cần reasoning mạnh → Haiku
|
|─ Trích Xuất Dữ Liệu (từ document có cấu trúc)
|   └─ Gemini 2.5 Flash (nhanh, rẻ, tốt cho trích xuất)
|       Nếu document > 200K token → Gemini 2.5 Pro (long context)
|
|─ Sinh Code
|   └─ Python, infrastructure, DevOps → Claude Sonnet 4.6
|       └─ JavaScript/TypeScript ecosystem → GPT-4o hoặc GPT-4.1
|           └─ Boilerplate đơn giản → Claude Haiku 4.5
|               └─ Thiết kế kiến trúc mới → Claude Opus 4.6
```

```

|
| └ Review Code
|   └ Claude Sonnet 4.6 (đáng tin cậy, cost-effective)
|     Review bảo mật critical → Claude Opus 4.6
|
| └ Tóm Tắt Tài Liệu
|   └ < 50K token → Claude Sonnet 4.6 hoặc Gemini 2.5 Flash
|   └ 50K–200K token → Gemini 2.5 Pro hoặc GPT-4.1
|     └ > 200K token → Gemini 2.5 Pro (long-context rẻ nhất)
|
| └ Reasoning / Giải Quyết Vấn Đề
|   └ Vấn đề rõ ràng, domain đã biết → Claude Sonnet 4.6
|   └ Vấn đề mơ hồ, domain mới → Claude Opus 4.6
|     └ Toán học / logic / formal reasoning → o3 hoặc o4-mini
|
| └ Sinh Config Infrastructure
|   └ Claude Sonnet 4.6 (tốt nhất cho các stack niche theo kinh nghiệm sách)
|     KHÔNG dùng Gemini Flash cho infra config (xem Chương 16, Lỗi #9)
|
| └ Phân Tích Alert / Log (bulk)
|   └ Gemini 2.5 Flash (pattern matching nhanh, chi phí thấp)
|     Batch API nếu không nhạy cảm về thời gian (tiết kiệm thêm 50%)
|
| └ Multimodal (ảnh, video, audio)
|   └ Ảnh + text → GPT-4o hoặc Gemini 2.5 Pro
|   └ Phân tích video → Gemini 2.5 Pro (hỗ trợ video native)
|   └ Transcription audio + phân tích → Gemini 2.5 Pro

```

CÂU HỎI THỨ HAI: Task có nhạy cảm về latency không?

Có → Tránh batch API; ưu tiên tier Flash/Haiku cho task không critical

Không → Cân nhắc batch API (tiết kiệm 50%, SLA 24h)

CÂU HỎI THỨ BA: Đây có phải task định kỳ/scheduled không?

Có → Áp dụng tiêu chí chi phí nghiêm ngặt hơn (xem Checklist Quản Lý Chi Phí)

Nếu giá trị không rõ ràng → xóa job đó (Chương 16, Lỗi #8)

Không → Tiến hành với model selection ở trên

Bảo Trì Checklist

Các checklist này là tài liệu sống. Sau mỗi sự cố hoặc deployment đáng kể, review và cập nhật checklist liên quan:

- Thêm bất kỳ bước chẩn đoán nào bạn phải tự tìm ra trong lúc xử lý
- Bỏ các bước không liên quan đến stack cụ thể của bạn
- Điều chỉnh ngưỡng cho phù hợp với thực tế vận hành (budget ceiling, alert threshold)
- Đánh dấu ngày tháng cho các lần revision lớn

Lưu phiên bản hiện tại của các checklist này trong git repository cùng với infrastructure code. Chạy `git log appendices/C-checklists.md` để xem lịch sử revision.

```
# Kiểm tra nhanh tình trạng checklist – verify các guard chính đang hoạt động
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)"
}
```

Hết Phụ lục C. Hết sách.