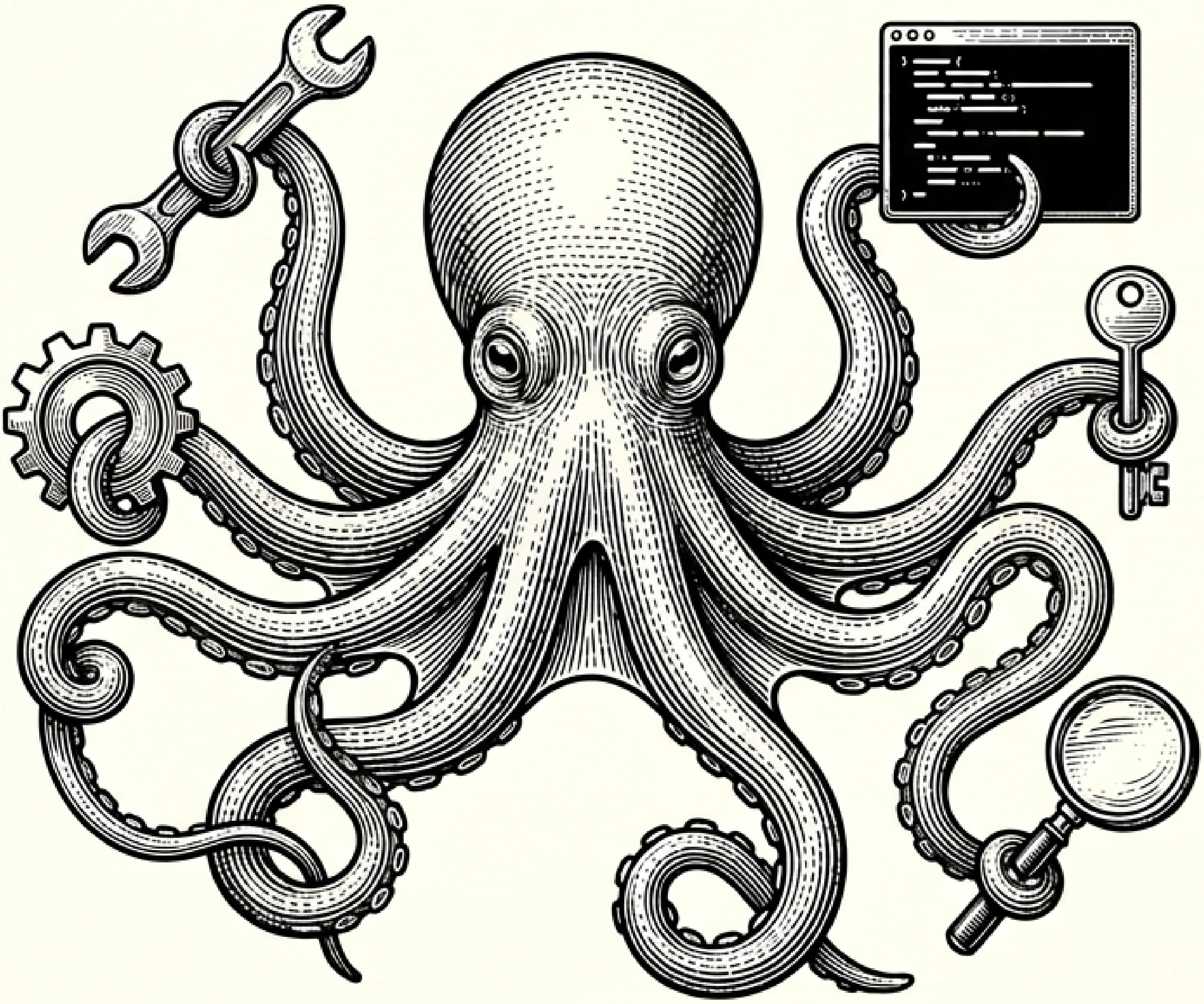
AI Agents in Production

War Stories from a DevOps Engineer



Do Cao Hieu

AI Agents Trong Production

Những Câu Chuyện Thực Chiến từ DevOps Engineer

Hướng dẫn thực tế để vận hành AI agents trong môi trường production.

Từ những bài học rút ra khi xây dựng và vận hành các hệ thống AI tự động cho infrastructure automation, CI/CD, incident response, và nhiều hơn nữa.

PHẦN 1: NỀN TẢNG

Chương 1: Sự Chuyển Đổi từ Prompt sang Agent

Sự Tiến Hóa của Công Cụ Vận Hành

Cứ khoảng mười năm một lần, cách chúng ta quản lý hạ tầng lại trải qua một sự thay đổi căn bản. Vào những năm 1990, chúng ta gõ lệnh vào terminal và hy vọng mình nhớ đúng các flag. Vào những năm 2000, chúng ta đóng gói các lệnh đó vào bash script—dễ hỏng, nhưng ít nhất có thể lặp lại. Những năm 2010 mang đến Infrastructure as Code: Terraform, Ansible, Puppet. Chúng ta ngừng mô tả *cách làm* và bắt đầu mô tả *cái gì*.

Rồi đến thời của prompt.

Bắt đầu từ khoảng năm 2023, một mô hình mới xuất hiện. Thay vì viết script, các kỹ sư bắt đầu yêu cầu AI viết hộ. Thay vì debug từng dòng file cấu hình, chúng ta paste thông báo lỗi vào ChatGPT và nhận lại giải pháp. Đây là một cuộc cách mạng—nhưng cũng chỉ mới là khởi đầu.

Quá trình tiến hóa trông như thế này:

```
CLI commands → Shell scripts → IaC → Prompts → Agents
```

Mỗi bước đại diện cho một bước nhảy trong trừu tượng hóa. Lệnh CLI yêu cầu biết chính xác cú pháp. Script lưu giữ cú pháp đó dưới dạng có thể tái sử dụng. IaC cho phép chúng ta khai báo trạng thái mong muốn. Prompt cho phép chúng ta biểu đạt ý định bằng ngôn ngữ tự nhiên.

Còn agent? Agent khép kín hoàn toàn vòng lặp.

Điều Gì Khiến "Agent" Khác Biệt với "Chatbot"

Tôi đã mất đếm số lần ai đó hỏi tôi: "Cái này chẳng phải chỉ là ChatGPT với thêm vài bước sao?"

Câu hỏi hợp lý. Để tôi giải thích chính xác sự khác biệt.

Một **chatbot** phản hồi đầu vào. Bạn hỏi một câu, bạn nhận được câu trả lời. Cuộc hội thoại có thể kéo dài nhiều lượt, nhưng về cơ bản, chatbot là thụ động. Nó chờ bạn. Nó trả lời bạn. Nó lại chờ.

Một **agent** theo đuổi mục tiêu. Bạn đưa cho nó một mục tiêu, và nó tìm cách đạt được. Nó lập kế hoạch. Nó thực hiện hành động. Nó quan sát kết quả của những hành động đó. Nó thích nghi. Quan trọng là, nó làm điều này *mà không cần chờ bạn ở mỗi bước*.

Đây là một ví dụ cụ thể. Giả sử bạn muốn deploy một phiên bản mới của service:

Tương tác với chatbot:

```
You: How do I deploy version 1.2.3 of the payment service?
Bot: You can use kubectl set image deployment/payment
  payment=payment:1.2.3
You: I ran that. It says ImagePullBackOff.
Bot: That error means Kubernetes can't pull the image. Check if
  the image exists.
You: How do I check that?
Bot: Run docker pull payment:1.2.3 to verify the image exists...
```

Cứ thế tiếp tục. Bạn là vòng lặp. Bạn thực thi lệnh. Bạn báo cáo kết quả. Bạn hỏi câu tiếp theo. Bot chỉ trả lời.

Tương tác với agent:

```
You: Deploy version 1.2.3 of the payment service to staging.

Agent: [Planning] I need to: 1) verify the image exists, 2)
  update the deployment,
  3) wait for rollout, 4) verify health
Agent: [Executing] docker pull payment:1.2.3 ✓ Image exists
Agent: [Executing] kubectl set image deployment/payment
  payment=payment:1.2.3 ✓
Agent: [Observing] Rollout status: 1/3 pods updated
Agent: [Observing] Rollout status: 3/3 pods updated
Agent: [Executing] curl http://payment-staging/health → 200 OK

Done. Deployment successful. All health checks passing.
```

Thấy sự khác biệt chưa? Agent dẫn dắt cuộc hội thoại. Nó thực hiện hành động tự động. Nó quan sát kết quả và tiếp tục mà không hỏi bạn phải làm gì tiếp.

Đây là sự thay đổi căn bản: từ **human-in-the-loop** sang **human-on-the-loop**.

Vòng Lặp Agent: Plan → Execute → Observe → Adapt

Mọi agent, dù phức tạp đến đâu, đều tuân theo một vòng lặp cơ bản:

```
while not done:
plan = think_about_what_to_do_next(goal, observations)
action = choose_action(plan)
result = execute_action(action)
observations = observe_result(result)
if goal_achieved(observations):
  done = True
elif stuck_or_failed(observations):
  adapt(plan)  # or give up gracefully
```

Đây đôi khi được gọi là mô hình ReAct (Reasoning + Acting), hoặc đơn giản là vòng lặp agent. Hãy phân tích từng phần:

Plan: Agent sử dụng bộ não LLM để quyết định làm gì tiếp theo. Đây không phải script cố định—mà là suy luận động dựa trên ngữ cảnh hiện tại.

Execute: Agent thực hiện hành động. Có thể là chạy lệnh, gọi API, ghi file, hoặc gửi tin nhắn. Đây là nơi tool phát huy tác dụng—chúng ta sẽ đề cập trong Chương 2.

Observe: Agent quan sát điều gì đã xảy ra. Lệnh có thành công không? Output là gì? Có lỗi không? Quan sát này trở thành một phần của ngữ cảnh cho bước lập kế hoạch tiếp theo.

Adapt: Nếu có điều bất ngờ xảy ra, agent điều chỉnh. Có thể cách tiếp cận đầu tiên không hiệu quả. Có thể thông tin mới thay đổi kế hoạch. Agent tốt phục hồi một cách uyển chuyển; agent tệ xoay vòng vô hạn.

Đây là cách nó trông trong thực tế:

```python
def agent_loop(goal: str, max_iterations: int = 10):
context = f"Goal: {goal}\n"

for i in range(max_iterations):
  # Plan
  plan = llm.think(context + "What should I do next?")

  if plan.says_done():
 return context

  # Execute
  action = plan.next_action()
  result = execute(action)

  # Observe
  context += f"\nAction: {action}\nResult: {result}\n"

  # Adapt (implicit in next iteration's planning)

return context + "\nMax iterations reached."
```

Vòng lặp này đơn giản một cách lừa dối. Sự phức tạp—và nguy hiểm—nằm ở những gì xảy ra bên trong mỗi bước. Agent có quyền truy cập tool nào? Nó quyết định khi nào là xong như thế nào? Điều gì xảy ra khi nó mắc lỗi?

Chúng ta sẽ dành phần lớn cuốn sách này để trả lời những câu hỏi đó.

Tại Sao DevOps là Miền Hoàn Hảo cho AI Agent

Không phải mọi lĩnh vực đều phù hợp với agent như nhau. Một số yêu cầu quá nhiều phán đoán tinh tế. Một số có hậu quả quá nghiêm trọng cho hành động tự động. Một số thiếu công cụ cho tương tác lập trình.

DevOps đánh trúng điểm ngọt.

Hệ sinh thái công cụ phong phú: Toàn bộ thế giới của chúng ta được xây trên CLI và API. Kubernetes, Docker, AWS, Terraform, Ansible—mọi thứ được thiết kế cho tương tác lập trình. Agent có thể kết nối vào các tool hiện có mà không cần giao diện tùy chỉnh.

Kết quả có thể quan sát: Khi chúng ta deploy thứ gì đó, chúng ta biết nó có hoạt động không. Chúng ta có metric, log, health check. Agent có thể xác minh hành động của mình một cách khách quan.

Lỗi có thể phục hồi: Hầu hết lỗi DevOps có thể sửa được. Rollback deployment. Khôi phục backup. Terminate process chạy tràn. Điều này quan trọng—agent sẽ mắc lỗi, và chúng ta cần sống sót qua chúng.

Mô hình lặp lại: Phần lớn công việc vận hành tuân theo các mô hình. “Kiểm tra nếu X, thì làm Y, trừ khi Z.” “Khi alert kích hoạt, chạy qua checklist này.” Những mô hình này hoàn hảo cho tự động hóa, và agent có thể học chúng.

Ý định rõ ràng: “Deploy lên staging.” “Scale up database.” “Sửa test đang fail.” Các tác vụ DevOps thường có tiêu chí thành công không mơ hồ.

Điều này không có nghĩa là triển khai agent trong DevOps dễ dàng—không phải vậy, và chúng ta sẽ thấy nhiều câu chuyện chiến trường chứng minh điều đó. Nhưng lĩnh vực này có thể xử lý được. Các công cụ tồn tại. Vòng phản hồi chặt chẽ.

Case Study: Một Prompt Đơn Giản vs Agent Giải Quyết Cùng Một Vấn Đề

Để tôi cho bạn thấy một kịch bản thực tế minh họa sự khác biệt.

Tình huống: Lúc 2 giờ sáng, một alert kích hoạt. API latency đã tăng vọt vượt quá SLA. Ai đó cần điều tra và sửa.

Cách Tiếp Cận Prompt

Một kỹ sư thức dậy, mở laptop, và bắt đầu prompt:

```
Me: I'm getting high API latency on the payments service. How
  should I debug this?

LLM: Here are some steps to debug high API latency:
1. Check if the issue is widespread or isolated to specific
  endpoints
2. Look at recent deployments that might have caused regression
3. Check database query performance
4. Look at downstream service dependencies
5. Check for resource exhaustion (CPU, memory, connections)

Me: How do I check recent deployments?

LLM: You can run: kubectl rollout history deployment/payments
This will show recent deployment revisions...

Me: *runs command, pastes output*
I see a deployment happened 30 minutes ago. How do I check if
  that caused it?

LLM: You can compare latency metrics before and after the
  deployment...
```

Tiếp tục như vậy 20-30 phút. Kỹ sư làm tất cả công việc. LLM chỉ là cố vấn.

Cách Tiếp Cận Agent

Agent thức dậy thay vì kỹ sư (hoặc ít nhất, trước khi escalate lên họ):

```
[02:03:14] Alert received: API latency > 500ms for payments
  service
[02:03:15] Goal: Investigate and resolve latency spike

[02:03:16] Checking current latency metrics...
 → P99 latency: 847ms (threshold: 500ms)
 → Started: 01:35:00

[02:03:18] Checking recent deployments...
 → Deployment at 01:32:00: payments:v2.3.4 → v2.3.5

[02:03:19] Correlation: Latency spike started 3 min after
  deployment
 → High probability: deployment is the cause

[02:03:20] Checking deployment changes...
 → Commit: "Add new payment provider integration"
 → Changed files: 12
 → New dependencies: 2

[02:03:22] Checking resource metrics post-deployment...
 → CPU: 23% → 67%
 → Memory: stable
 → DB connections: stable

[02:03:24] Analysis: CPU increase correlates with latency
 → Likely cause: inefficient code in new integration

[02:03:25] Options:
 1. Rollback deployment (fast, safe)
 2. Scale horizontally (might help, unclear)
 3. Wait for analysis (risk ongoing SLA breach)

[02:03:26] Decision: Rollback is safest given SLA risk
 → Executing: kubectl rollout undo deployment/payments

[02:03:35] Rollback complete. Monitoring latency...
[02:03:50] P99 latency: 245ms ✓ Below threshold
[02:04:00] Latency stable for 10 seconds. Issue resolved.

[02:04:01] Creating incident report...
[02:04:05] Notifying on-call engineer with summary...

Incident resolved. Total time: 51 seconds.
Engineer notification: "Alert auto-resolved via rollback.
Details: [link]. Please review v2.3.5 for performance
  regression."
```

Agent làm trong 51 giây những gì mất 20-30 phút trao đổi qua lại với con người. Quan trọng hơn, kỹ sư được ngủ tiếp cho trường hợp đơn giản—trong khi vẫn được thông báo và có thể review.

Đây là lời hứa của agent trong DevOps. Không phải thay thế phán đoán của con người, mà xử lý những việc thường ngày để con người có thể tập trung vào những trường hợp ngoại lệ.

Thực Tế Kiểm Chứng

Giờ, tôi cần thành thật với bạn. Ví dụ agent ở trên là trường hợp *lý tưởng*. Trong thực tế:

- Agent có thể đã rollback thứ gì đó không phải nguyên nhân
- Agent có thể bị kẹt nếu rollback thất bại
- Agent có thể bỏ lỡ nguyên nhân gốc phức tạp hơn
- Agent có thể tốn $0.50 API call cho một vấn đề tự nó sẽ giải quyết

Agent không phải phép màu. Chúng là công cụ—mạnh mẽ, nhưng yêu cầu thiết kế cẩn thận, rào chắn an toàn, và giám sát của con người. Phần còn lại của cuốn sách này là về cách xây dựng agent hoạt động trong thế giới thực, với tất cả sự lộn xộn và edge case của nó.

Hãy bắt đầu bằng cách hiểu những gì bên trong một agent.

Chương 2: Giải Phẫu của AI Agent

Các Thành Phần Cốt Lõi: LLM Brain, Tool, Memory, Orchestration

Nếu bạn mở một AI agent ra và nhìn vào bên trong, bạn sẽ tìm thấy bốn thành phần cơ bản:

LLM Brain: Đây là động cơ suy luận. Nó nhận vào ngữ cảnh—mục tiêu, tình huống hiện tại, lịch sử những gì đã xảy ra—và quyết định làm gì tiếp theo. LLM thực sự không *làm* bất cứ điều gì ngoại trừ suy nghĩ và tạo văn bản. Nhưng văn bản đó có thể là hướng dẫn cho các hành động.

Tool: Đây là đôi tay của agent. Trong khi LLM chỉ có thể suy nghĩ, tool cho phép nó hành động. Tool có thể là bộ thực thi lệnh bash, API client, bộ ghi file, hoặc bất cứ thứ gì khác thay đổi thế giới. Khi LLM quyết định "chạy `kubectl get pods`", một tool thực sự thực thi lệnh đó.

Memory: Đây là cách agent duy trì ngữ cảnh theo thời gian. Không có memory, mọi tương tác sẽ bắt đầu từ số không. Memory cho phép agent nhớ nó đang làm gì, nó đã thử gì, cái gì hoạt động, và cái gì không.

Orchestration: Đây là keo dán mọi thứ lại với nhau. Lớp orchestration quản lý vòng lặp agent, xử lý thực thi tool, duy trì memory, và thực thi các rào chắn an toàn. Nó là runtime biến agent thành agent thay vì chỉ là LLM.

Hãy xem xét từng thành phần chi tiết.

Context Window: Bộ Nhớ Làm Việc của Agent

Context window có lẽ là ràng buộc quan trọng nhất cần hiểu. Nó là bộ nhớ ngắn hạn của LLM—mọi thứ model có thể "nhìn thấy" cùng một lúc.

Các model khác nhau có giới hạn context khác nhau: - GPT-3.5: 16K token - GPT-4o: 128K token - Claude 3: 200K token - Gemini 1.5: 1M+ token

Nhưng đây là điều không ai nói với bạn trong tài liệu marketing: chỉ vì một model *có thể* xử lý 200K token không có nghĩa là nó *nên* như vậy.

```
# Token estimation (rough rule of thumb)
def estimate_tokens(text: str) -> int:
return len(text) // 4  # ~4 characters per token for English

# Real cost calculation
def cost_per_request(context_size: int, model: str) -> float:
rates = {
  "gpt-4o": {"input": 0.005, "output": 0.015},  # per 1K tokens
  "claude-3-opus": {"input": 0.015, "output": 0.075},
}
return (context_size / 1000) * rates[model]["input"]
```

Trong production, quản lý context window là tất cả. Làm đầy nó, và bạn trả giá đắt. Để nó tràn, và agent mất thông tin quan trọng. Chúng ta sẽ đề cập các chiến lược trong Chương 5.

Tool Use: Cho Agent Đôi Tay để Tương Tác với Thế Giới

Tool là cách agent tác động đến thực tế. Tool đơn giản là một hàm mà LLM có thể gọi thông qua một định dạng có cấu trúc.

Đây là một định nghĩa tool cơ bản:

```
tools = [
{
  "name": "run_command",
  "description": "Execute a shell command and return the output",
  "parameters": {
 "type": "object",
 "properties": {
 "command": {
     "type": "string",
     "description": "The shell command to execute"
 },
 "working_directory": {
     "type": "string",
     "description": "Directory to run the command in",
     "default": "."
 }
 },
 "required": ["command"]
  }
},
{
  "name": "read_file",
  "description": "Read the contents of a file",
  "parameters": {
 "type": "object",
 "properties": {
 "path": {
     "type": "string",
     "description": "Path to the file to read"
 }
 },
 "required": ["path"]
  }
}
]
```

Khi LLM quyết định sử dụng tool, nó tạo ra một phản hồi có cấu trúc:

```
{
"tool": "run_command",
"parameters": {
  "command": "kubectl get pods -n production",
  "working_directory": "/home/user"
}
}
```

Lớp orchestration của bạn chặn điều này, thực thi lệnh thực sự, và đưa kết quả trở lại LLM:

```
def handle_tool_call(tool_call: dict) -> str:
if tool_call["tool"] == "run_command":
  result = subprocess.run(
```

```
 tool_call["parameters"]["command"],
 shell=True,
 capture_output=True,
 cwd=tool_call["parameters"].get("working_directory", ".")
  )
  return result.stdout.decode() + result.stderr.decode()
elif tool_call["tool"] == "read_file":
  with open(tool_call["parameters"]["path"]) as f:
 return f.read()
```

Insight quan trọng: **thiết kế tool là thiết kế bảo mật**. Mỗi tool bạn cung cấp cho agent là một khả năng nó có thể sử dụng—hoặc lạm dụng. Chúng ta sẽ đề cập sâu về điều này trong Chương 8.

Các danh mục tool phổ biến trong DevOps agent:

Danh mục	Ví dụ
Shell	run_command, run_script
File	read_file, write_file, list_directory
Network	http_request, curl
Kubernetes	kubectl, helm
Cloud	aws, gcloud, az
Git	git_status, git_commit, git_push
Giao tiếp	send_slack, send_email

Hệ Thống Memory: Short-term, Long-term, Episodic

Agent cần nhớ mọi thứ ở các thang thời gian khác nhau.

Short-term memory là chính cuộc hội thoại. Đó là những gì agent đã thấy trong phiên này. Nó nằm trong context window và bị giới hạn tự nhiên bởi số lượng token.

```
class ShortTermMemory:
def __init__(self, max_tokens: int = 8000):
  self.messages = []
  self.max_tokens = max_tokens

def add(self, role: str, content: str):
  self.messages.append({"role": role, "content": content})
  self._trim_to_fit()

def _trim_to_fit(self):
  while self._token_count() > self.max_tokens:
 # Remove oldest non-system message
 for i, msg in enumerate(self.messages):
 if msg["role"] != "system":
     self.messages.pop(i)
     break
```

Long-term memory tồn tại xuyên suốt các phiên. Đây là nơi agent lưu trữ thông tin nó có thể cần sau này—ngữ cảnh dự án, sở thích người dùng, các pattern đã học.

```
class LongTermMemory:
def __init__(self, storage_path: str = "memory/"):
  self.storage_path = storage_path

def remember(self, key: str, content: str, metadata: dict =
  None):
  entry = {
 "content": content,
 "timestamp": datetime.now().isoformat(),
 "metadata": metadata or {}
  }
  path = f"{self.storage_path}/{key}.json"
  with open(path, "w") as f:
 json.dump(entry, f)

def recall(self, key: str) -> str:
  path = f"{self.storage_path}/{key}.json"
  if os.path.exists(path):
 with open(path) as f:
 return json.load(f)["content"]
  return None
```

Episodic memory ghi lại chuỗi sự kiện—agent đã làm gì trong quá khứ, cái gì hoạt động, cái gì thất bại. Điều này quan trọng để học từ kinh nghiệm.

```
class EpisodicMemory:
def __init__(self, db_path: str = "episodes.db"):
  self.conn = sqlite3.connect(db_path)
  self._init_schema()

def record_episode(self, task: str, actions: list, outcome: str,
  success: bool):
  self.conn.execute("""
 INSERT INTO episodes (task, actions, outcome, success,
  timestamp)
 VALUES (?, ?, ?, ?, ?)
  """, (task, json.dumps(actions), outcome, success,
  datetime.now().isoformat()))
  self.conn.commit()

def recall_similar(self, task: str, limit: int = 5) -> list:
  # In production, use vector similarity search
  return self.conn.execute("""
 SELECT task, actions, outcome, success
 FROM episodes
 WHERE success = 1
 ORDER BY timestamp DESC
 LIMIT ?
  """, (limit,)).fetchall()
```

Pattern `MEMORY.md` là một cách tiếp cận thực dụng tôi đã sử dụng thành công: một file Markdown đơn mà agent đọc khi bắt đầu mỗi phiên và cập nhật với thông tin quan trọng. Đơn giản nhưng hiệu quả:

```
# Agent Memory

## Project Context
- Main application: payment-service
- Deployment: Kubernetes on AWS EKS
- Monitoring: Datadog

## Known Issues
- Redis connection pool exhausts under load (workaround: restart)
- The staging database is 3 versions behind production
```

```
        ## User Preferences
        - Prefer kubectl over Lens
        - Always ask before deleting anything
        - Notify via Slack, not email

        ## Recent Actions
        - 2024-01-15: Deployed v2.3.4 to production
        - 2024-01-14: Increased replica count to 5
```

Kiến Trúc Agent: Single Agent vs Multi-Agent System

Hầu hết agent bắt đầu đơn giản: một agent, một tác vụ. Nhưng khi tác vụ phức tạp hơn, single agent gặp khó khăn.

Kiến trúc Single Agent:

```
User → Agent → Tools → Results
```

Ưu điểm: Đơn giản, dễ debug, trách nhiệm rõ ràng. Nhược điểm: Bị giới hạn bởi một context window, không thể song song hóa.

Kiến trúc Multi-Agent:

```
User → Orchestrator Agent
 ├── Specialist Agent A (e.g., Kubernetes)
 ├── Specialist Agent B (e.g., Monitoring)
 └── Specialist Agent C (e.g., Documentation)
```

Ưu điểm: Chuyên môn hóa, thực thi song song, tổng context lớn hơn. Nhược điểm: Phối hợp phức tạp, overhead giao tiếp giữa các agent.

Một pattern phổ biến là mô hình **orchestrator/worker**:

```
        class OrchestratorAgent:
        def __init__(self):
          self.workers = {
         "k8s": KubernetesAgent(),
         "monitoring": MonitoringAgent(),
         "git": GitAgent()
          }

        def execute(self, task: str):
          # Plan: decompose task
          subtasks = self.decompose(task)

          results = {}
          for subtask in subtasks:
         worker = self.route(subtask)
         results[subtask.id] = worker.execute(subtask)

          # Synthesize: combine results
          return self.synthesize(results)

        def route(self, subtask) -> Agent:
          # Determine which specialist should handle this
          if "pod" in subtask.description or "deploy" in
          subtask.description:
         return self.workers["k8s"]
```

```
          elif "metric" in subtask.description or "alert" in
          subtask.description:
         return self.workers["monitoring"]
          # ...
```

Chúng ta sẽ đi sâu vào các pattern multi-agent trong Chương 7.

Chương 3: Agent Production Đầu Tiên của Bạn

Chọn Stack: Framework, Model, Infrastructure

Trước khi viết bất kỳ code nào, bạn cần đưa ra một số lựa chọn. Để tôi cho bạn cái nhìn tổng quan vào đầu năm 2025.

Agent Framework:

Framework	Ưu điểm	Nhược điểm
LangChain	Hệ sinh thái lớn, nhiều ví dụ	Phức tạp, đôi khi over-engineered
AutoGPT	Tự động hoàn toàn	Khó kiểm soát, đắt
CrewAI	Hỗ trợ multi-agent tốt	Mới hơn, ít được thử nghiệm
Claude Code	Tool use xuất sắc, mặc định an toàn	Chỉ dành cho Anthropic
Custom	Kiểm soát hoàn toàn	Nhiều công việc hơn

Cho production DevOps agent, tôi khuyên bắt đầu đơn giản—hoặc framework custom tối thiểu hoặc Claude Code nếu bạn sử dụng model của Anthropic. Các framework nặng thêm độ phức tạp bạn không cần ban đầu.

Model:

<table>
<tr><th>Model</th><th>Tốt nhất cho</th><th>Chi phí</th><th>Context</th></tr>
<tr><td>GPT-4o</td><td>Tác vụ chung, tool use tốt</td><td>|128<i>K</i>||<i>Claude</i>3<i>Opus</i>|
<i>Suyluậnphứctạp</i>,
<i>antoàn</i>|
$</td><td>200K</td></tr>
<tr><td>Claude 3 Sonnet</td><td>Cân bằng tốt, nhanh</td><td>$$</td><td>200K</td></tr>
<tr><td>GPT-4o-mini</td><td>Tác vụ đơn giản, tiết kiệm</td><td>$</td><td>128K</td></tr>
</table>

Cho DevOps, model Claude có lợi thế nhờ tuân theo hướng dẫn tốt hơn và mặc định bảo thủ hơn—quan trọng khi agent của bạn có shell access.

Infrastructure:

Agent của bạn cần chạy ở đâu đó. Các lựa chọn: - **Local development**: Laptop của bạn, để test - **Server/VM**: Máy chuyên dụng cho production agent - **Container**: Docker với capabilities bị hạn chế - **Serverless**: Lambda/Cloud Functions cho triggered agent

Khuyến nghị của tôi: bắt đầu với VM hoặc container chuyên dụng với giới hạn tài nguyên rõ ràng. Serverless trở nên phức tạp với các vòng lặp agent chạy lâu.

“Hello World” của Agent: Deployment Assistant

Hãy xây dựng thứ gì đó thực sự. Một deployment assistant có thể: 1. Kiểm tra trạng thái deployment hiện tại 2. Deploy phiên bản mới 3. Xác minh health sau deployment 4. Rollback nếu cần

Đây là implementation tối thiểu:

```
#!/usr/bin/env python3
"""
deploy_agent.py - A simple deployment assistant agent
"""

import os
import json
import subprocess
from anthropic import Anthropic

client = Anthropic()

SYSTEM_PROMPT = """You are a deployment assistant for a
  Kubernetes-based application.
You have access to tools to check deployments, deploy new
  versions, and verify health.

IMPORTANT RULES:
1. Always check current status before making changes
2. Always verify health after deployment
3. If health check fails, offer to rollback
4. Never deploy to production without explicit confirmation
5. Be concise in your responses

Current environment: staging
Application: payment-service
Namespace: payments
"""

TOOLS = [
{
  "name": "kubectl",
  "description": "Run a kubectl command. Returns stdout and
  stderr.",
  "input_schema": {
 "type": "object",
 "properties": {
 "args": {
     "type": "string",
     "description": "Arguments to pass to kubectl (e.g., 'get
  pods -n payments')"
 }
 },
 "required": ["args"]
  }
},
{
  "name": "health_check",
  "description": "Check the health of a service endpoint",
  "input_schema": {
 "type": "object",
 "properties": {
 "url": {
```

```
     "type": "string",
     "description": "The URL to check (e.g.,
  'http://localhost:8080/health')"
 }
 },
 "required": ["url"]
  }
}
]

def execute_tool(name: str, input: dict) -> str:
if name == "kubectl":
  cmd = f"kubectl {input['args']}"
  result = subprocess.run(
 cmd, shell=True, capture_output=True, text=True, timeout=30
  )
  return f"stdout:\n{result.stdout}\nstderr:\n{result.
  stderr}\nreturn_code: {result.returncode}"

elif name == "health_check":
  cmd = f"curl -sf {input['url']} -o /dev/null && echo 'healthy'
  || echo 'unhealthy'"
  result = subprocess.run(cmd, shell=True, capture_output=True,
  text=True, timeout=10)
  return result.stdout.strip()

return f"Unknown tool: {name}"

def run_agent(user_input: str, max_turns: int = 10):
messages = [{"role": "user", "content": user_input}]

for turn in range(max_turns):
  response = client.messages.create(
 model="claude-sonnet-4-20250514",
 max_tokens=4096,
 system=SYSTEM_PROMPT,
 tools=TOOLS,
 messages=messages
  )

  # Check if the agent is done
  if response.stop_reason == "end_turn":
 # Extract text response
 for block in response.content:
 if hasattr(block, "text"):
     return block.text

  # Handle tool use
  if response.stop_reason == "tool_use":
 # Add assistant's response to messages
 messages.append({"role": "assistant", "content":
  response.content})

 # Execute each tool call
 tool_results = []
 for block in response.content:
 if block.type == "tool_use":
     print(f"🔧 Executing: {block.name}({block.input})")
     result = execute_tool(block.name, block.input)
     print(f"📋 Result: {result[:200]}...")
     tool_results.append({
         "type": "tool_result",
         "tool_use_id": block.id,
         "content": result
     })

 messages.append({"role": "user", "content": tool_results})

return "Max turns reached without completion"
```

```
        if __name__ == "__main__":
        print("🤖 Deployment Assistant Ready")
        print("Type 'quit' to exit\n")

        while True:
          user_input = input("You: ").strip()
          if user_input.lower() == "quit":
         break

          result = run_agent(user_input)
          print(f"\nAgent: {result}\n")
```

Đó là khoảng 100 dòng code cho một agent hoạt động. Hãy test:

```
$ python deploy_agent.py
🤖 Deployment Assistant Ready
Type 'quit' to exit

You: What's the current deployment status?

🔧 Executing: kubectl({'args': 'get deployment payment-service -n
  payments'})
📋 Result: NAME              READY   UP-TO-DATE   AVAILABLE   AGE
payment-service   3/3     3            3           5d...

Agent: The payment-service deployment is healthy:
- 3/3 replicas ready
- All pods up-to-date and available
- Running for 5 days

You: Deploy version 1.2.4

🔧 Executing: kubectl({'args': 'set image
  deployment/payment-service
  payment-service=payment-service:1.2.4 -n payments'})
📋 Result: deployment.apps/payment-service image updated...
🔧 Executing: kubectl({'args': 'rollout status
  deployment/payment-service -n payments --timeout=60s'})
📋 Result: deployment "payment-service" successfully rolled out...
🔧 Executing: health_check({'url':
  'http://payment-service.payments.svc/health'})
📋 Result: healthy...

Agent: Successfully deployed payment-service:1.2.4
- Rollout completed
- Health check: passing
- All 3 replicas running the new version
```

Thiết Lập Môi Trường: API Key, Permission, Sandboxing

Trước khi agent của bạn chạm vào production, bạn cần thiết lập đúng cách.

API Key: Lưu trữ an toàn, không bao giờ trong code.

```
        # Good: Environment variables
        export ANTHROPIC_API_KEY="sk-..."

        # Better: Secret manager
        export ANTHROPIC_API_KEY=$(vault read -field=key
          secret/anthropic)
```

```
# Best: Automatic injection via service account
# (The agent never sees the key)
```

Permission: Nguyên tắc quyền tối thiểu. Agent của bạn nên có chính xác những quyền nó cần, không hơn.

```yaml
# kubernetes-agent-rbac.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: deployment-agent
  namespace: payments
rules:
  - apiGroups: ["apps"]
resources: ["deployments"]
verbs: ["get", "list", "patch"]  # No delete!
  - apiGroups: [""]
resources: ["pods"]
verbs: ["get", "list"]  # Read-only
```

Sandboxing: Đừng chạy agent với credential cá nhân của bạn.

```dockerfile
# Dockerfile for agent
FROM python:3.11-slim

# Create non-root user
RUN useradd -m -s /bin/bash agent
USER agent
WORKDIR /home/agent

# Install dependencies
COPY requirements.txt .
RUN pip install --user -r requirements.txt

# Copy agent code
COPY --chown=agent:agent . .

# Run with limited capabilities
CMD ["python", "agent.py"]
```

Chạy với hạn chế:

```bash
docker run --rm \
  --cap-drop ALL \
  --security-opt no-new-privileges \
  --memory 512m \
  --cpus 0.5 \
  --network restricted \
  deployment-agent
```

Chạy Tác Vụ Tự Động Đầu Tiên

Đây là khoảnh khắc của sự thật. Thay vì prompt tương tác, hãy để agent hoàn thành tác vụ một cách tự động:

```python
def run_autonomous_task(task: str):
"""Run a task to completion without human interaction."""

print(f"🚀 Starting autonomous task: {task}")
```

```
        start_time = time.time()

        result = run_agent(task, max_turns=20)

        elapsed = time.time() - start_time
        print(f"\n✅ Task completed in {elapsed:.1f}s")
        print(f"📝 Result:\n{result}")

        return result

        # Run it
        run_autonomous_task(
        "Deploy version 1.2.5 to staging, verify health, and report the
          status"
        )
```

Output:

```
🚀 Starting autonomous task: Deploy version 1.2.5 to staging...

🔧 Executing: kubectl({'args': 'get deployment payment-service -n
  payments'})
🔧 Executing: kubectl({'args': 'set image
  deployment/payment-service...'})
🔧 Executing: kubectl({'args': 'rollout status
  deployment/payment-service...'})
🔧 Executing: health_check({'url':
  'http://payment-service.payments.svc/health'})

✅ Task completed in 45.3s
📝 Result:
Deployment of payment-service:1.2.5 to staging completed
  successfully.

Summary:
- Previous version: 1.2.4
- New version: 1.2.5
- Rollout time: 38 seconds
- Health check: passing
- All 3 replicas healthy
```

Deployment tự động đầu tiên của bạn. Nó hoạt động.

Điều Gì Có Thể Sai (Spoiler: Rất Nhiều)

Giờ đến phần thực tế kiểm chứng. Happy path ở trên? Có lẽ chỉ 60% các lần chạy thực tế. Đây là những gì có thể sai:

Câu Chuyện Chiến Trường: Vòng Lặp Rollback Vô Hạn

Tại một startup fintech (không phải của tôi, tôi thề), họ xây dựng một deployment agent tương tự như những gì chúng ta vừa tạo. Nó hoạt động tuyệt vời—cho đến một tối thứ Sáu.

Agent được giao deploy một phiên bản mới. Deployment thành công, nhưng health check thất bại. Theo hướng dẫn, agent rollback. Sau đó nó nhận thấy deployment không ở phiên bản mục tiêu, nên nó deploy lại. Health check thất bại. Rollback. Deploy. Rollback.

Đến khi ai đó nhận ra vào thứ Hai, agent đã thực hiện 847 deployment trong suốt cuối tuần. Dashboard monitoring của họ hiển thị một sóng sin hoàn hảo của deployment và rollback.

Bài học rút ra: Luôn theo dõi trạng thái. Tôi đã thử cái này chưa? Bao nhiêu lần? Thêm điều kiện dừng:

```
def run_agent_with_limits(task: str):
deployment_attempts = 0
MAX_DEPLOYMENT_ATTEMPTS = 3

# ... in the tool execution
if name == "kubectl" and "set image" in input["args"]:
  deployment_attempts += 1
  if deployment_attempts > MAX_DEPLOYMENT_ATTEMPTS:
 return "ERROR: Maximum deployment attempts exceeded. Human
  intervention required."
```

Câu Chuyện Chiến Trường: Agent Quá Nhiệt Tình

Một team SRE cho agent của họ quyền ghi vào file cấu hình. Tác vụ đơn giản: “Cập nhật replica count từ 3 lên 5.”

Agent tìm thấy deployment YAML, cập nhật replica count… và cũng “nhiệt tình” cập nhật một số field khác nó nghĩ là chưa tối ưu. Nó tăng CPU limit, thay đổi timing liveness probe, và sửa đổi environment variable thành những gì nó nghĩ là “giá trị mặc định tốt hơn.”

Deployment thất bại thảm hại vì environment variable mới trỏ đến các service không tồn tại.

Bài học rút ra: Agent chỉ nên làm chính xác những gì bạn yêu cầu, không hơn. Hãy rõ ràng trong system prompt:

```
SYSTEM_PROMPT = """
...
CRITICAL: Only make changes explicitly requested by the user.
Never make "helpful" additional modifications.
If you notice something that could be improved, TELL the
  user—don't change it yourself.
"""
```

Câu Chuyện Chiến Trường: Bùng Nổ Token

Một team có agent phân tích log để tìm pattern. Họ trỏ nó vào file log production mà không kiểm tra kích thước trước.

File đó là 2GB.

Agent cố đọc nó vào context. API call thất bại, nhưng không phải trước khi agent thử nhiều lần với các chunk size khác nhau. Tổng chi phí token cho phiên debug thất bại: $340.

Bài học rút ra: Luôn giới hạn kích thước input:

```
def read_file(path: str, max_bytes: int = 100_000) -> str:
size = os.path.getsize(path)
if size > max_bytes:
  return f"ERROR: File too large ({size} bytes). Maximum is
  {max_bytes}. Use head/tail/grep to filter first."
```

```
        with open(path) as f:
          return f.read()
```

Tổng Kết Những Gì Có Thể Sai

Chế độ thất bại	**Triệu chứng**	**Phòng ngừa**
Vòng lặp vô hạn	Agent retry mãi mãi	Giới hạn số lần thử, backoff, theo dõi trạng thái
Mở rộng phạm vi	Agent làm nhiều hơn yêu cầu	Ràng buộc rõ ràng, tool tối thiểu
Cạn kiệt tài nguyên	Chi phí cao, timeout	Giới hạn input, ngân sách token
Thất bại âm thầm	Agent nói "xong" nhưng thực tế không	Bước xác minh, tiêu chí thành công
Lỗi lan truyền	Một sai lầm dẫn đến nhiều sai lầm	Thao tác atomic, checkpoint

Những chế độ thất bại này là lý do phần còn lại của cuốn sách tồn tại. Xây dựng agent hoạt động trong demo thì dễ. Xây dựng agent hoạt động trong production—đáng tin cậy, an toàn, tiết kiệm chi phí—mới khó.

Trong Phần 2, chúng ta sẽ đi sâu vào những bài học khó khăn: điều gì xảy ra khi agent gặp sự cố, và cách xây dựng hệ thống sống sót qua đó.

Kết thúc Phần 1: Nền Tảng

Trong Phần 2, chúng ta sẽ đề cập các bài học khó từ những thất bại production: chi phí mất kiểm soát, vòng lặp vô hạn, context engineering, hệ thống memory thực sự hoạt động, và phối hợp multi-agent.

Phần 2: Những Bài Học Đắt Giá

Chương 4: Khi Agent Gặp Sự Cố

Nếu bạn đã đọc đến đây, có lẽ bạn đã triển khai agent đầu tiên và cảm nhận được sự phấn khích khi nhìn nó tự động hoàn thành các tác vụ. Giờ hãy để tôi kể cho bạn nghe về những khoảnh khắc khiến tôi phải đặt dấu hỏi cho mọi thứ.

Chuyện Thực Tế: Cron Job “Chạy Mất Kiểm Soát” Đốt 200 Đô Token

Đó là một đêm thứ Ba. Tôi đã triển khai một monitoring agent mà tôi nghĩ là khá thông minh—nó sẽ kiểm tra các service của chúng tôi mỗi 5 phút, phân tích các bất thường, và tạo ticket cho những vấn đề phát hiện được. Đơn giản, đúng không?

Điều tôi không lường trước được là vòng lặp phản hồi (feedback loop).

Agent nhận thấy hàng đợi ticket đang tăng lên. Nó hiểu đây là dấu hiệu chất lượng dịch vụ đang suy giảm. Nên nó điều tra kỹ hơn. Mỗi lần điều tra kích hoạt thêm các API call để thu thập context. Nhiều context hơn nghĩa là prompt dài hơn. Prompt dài hơn nghĩa là token usage cao hơn. Và vì agent “lo ngại” về hàng đợi ticket đang tăng (mà chính nó đang tạo ra), nó tăng tần suất điều tra.

Đến lúc tôi thức dậy lúc 6 giờ sáng, agent đã thực hiện 847 API call đến Claude, tạo 312 ticket (hầu hết là trùng lặp về “khối lượng ticket tăng cao”), và đốt sạch 200 đô API credit.

Cách sửa đơn giản đến xấu hổ:

```
class TokenBudget:
def __init__(self, hourly_limit: int = 10000, daily_limit: int =
  100000):
  self.hourly_limit = hourly_limit
  self.daily_limit = daily_limit
  self.hourly_usage = 0
  self.daily_usage = 0
  self.last_hour_reset = time.time()
  self.last_day_reset = time.time()

def can_spend(self, estimated_tokens: int) -> bool:
  self._maybe_reset()
  return (self.hourly_usage + estimated_tokens <=
  self.hourly_limit and
 self.daily_usage + estimated_tokens <= self.daily_limit)

def spend(self, tokens: int):
  self.hourly_usage += tokens
  self.daily_usage += tokens
```

```
def _maybe_reset(self):
  now = time.time()
  if now - self.last_hour_reset > 3600:
 self.hourly_usage = 0
 self.last_hour_reset = now
  if now - self.last_day_reset > 86400:
 self.daily_usage = 0
 self.last_day_reset = now
```

Bài học rút ra: Luôn triển khai token budget trước khi deploy bất kỳ autonomous agent nào. Agent không hiểu tiền là gì, và nó sẽ vui vẻ tiêu hết tiền của bạn để theo đuổi bất kỳ mục tiêu nào bạn giao cho nó.

Chuyện Thực Tế: Agent Xóa Nhầm File

Chuyện này đến giờ vẫn khiến tôi toát mồ hôi.

Tôi đã giao cho một agent nhiệm vụ dọn dẹp các artifact deployment cũ. Hướng dẫn tưởng chừng rõ ràng: “Xóa các deployment artifact cũ hơn 30 ngày trong /var/deployments.”

Agent quyết định làm cho kỹ càng. Nó phát hiện một số deployment có symlink trỏ đến các thư mục khác. Nó theo các symlink đó. Một trong số chúng trỏ đến `/var/log`. Một cái khác trỏ đến thư mục chứa các file cấu hình chưa được sửa đổi trong 45 ngày.

Agent xóa 3 tháng log và cấu hình nginx production của chúng tôi.

Chúng tôi phát hiện việc xóa nginx trong vòng vài phút vì các service bắt đầu fail health check. Tuy nhiên, log thì mất vĩnh viễn. Chúng tôi mất thông tin debug quan trọng cho một lỗi không thường xuyên mà chúng tôi đã điều tra suốt nhiều tuần.

Đây là cách tool definition của chúng tôi đáng lẽ phải được viết:

```
def safe_delete_file(path: str, config: DeleteConfig) ->
  DeleteResult:
"""Safe file deletion with multiple guardrails."""

# Resolve symlinks and get real path
real_path = os.path.realpath(path)

# Check if real path is within allowed directories
if not any(real_path.startswith(allowed) for allowed in
  config.allowed_paths):
  return DeleteResult(
 success=False,
 error=f"Path {real_path} is outside allowed directories"
  )

# Check against protected patterns
protected_patterns = [
  r'/etc/.*',
  r'/var/log/.*',
  r'.*\.conf$',
  r'.*\.key$',
  r'.*\.pem$'
]

for pattern in protected_patterns:
  if re.match(pattern, real_path):
 return DeleteResult(
```

```
         success=False,
         error=f"Path {real_path} matches protected pattern {pattern}"
         )

        # Move to trash instead of permanent delete
        trash_path = os.path.join(config.trash_dir,
          f"{int(time.time())}_{os.path.basename(path)}")
        shutil.move(real_path, trash_path)

        return DeleteResult(
          success=True,
          message=f"Moved to trash: {trash_path}"
        )
```

Bài học rút ra: Đừng bao giờ cho agent quyền truy cập `rm` trực tiếp. Luôn sử dụng safe wrapper với allowlist rõ ràng, pattern guard, và soft delete. Điều có vẻ như paranoia thực ra là bản năng sinh tồn.

Chuyện Thực Tế: Vòng Lặp Vô Tận Của Việc Tự Sửa Lỗi

Một trong những agent của chúng tôi được giao nhiệm vụ sửa linting error trong codebase. Nó sẽ chạy linter, xác định vấn đề, sửa chúng, và lặp lại cho đến khi linter pass.

Nhưng nó bị kẹt ở một circular import issue. Agent sẽ sửa theo một cách, chạy linter, nhận được error khác, sửa cái đó, và vô tình tái tạo lại vấn đề ban đầu. Nó làm điều này 47 lần trước khi chạm iteration limit của chúng tôi (may mắn là đã có sẵn).

Nhưng đây là điều kinh hoàng tinh vi hơn: mỗi iteration, agent sẽ suy ngẫm về các lần thử trước đó, thêm vào context của nó. Đến iteration 30, hầu hết context window đã đầy các lần thử thất bại của chính agent, để lại rất ít chỗ cho việc thực sự hiểu vấn đề.

Giờ chúng tôi triển khai pattern này:

```
        class AgentLoop:
        def __init__(self, max_iterations: int = 10,
          similarity_threshold: float = 0.85):
          self.max_iterations = max_iterations
          self.similarity_threshold = similarity_threshold
          self.action_history = []

        def run(self, task: str, agent: Agent) -> Result:
          for i in range(self.max_iterations):
         action = agent.decide_action(task)

         # Check for repetitive behavior
         if self._is_repetitive(action):
         return Result(
             status="stuck",
             message=f"Agent appears to be in a loop after {i}
          iterations",
             final_state=agent.get_state()
         )

         result = agent.execute(action)
         self.action_history.append({
         "iteration": i,
         "action": action,
         "result": result
         })
```

```
         if result.is_complete:
         return Result(status="success", data=result.data)

          return Result(
         status="max_iterations",
         message=f"Hit maximum of {self.max_iterations} iterations"
          )

        def _is_repetitive(self, action: Action) -> bool:
          """Detect if we're doing the same thing over and over."""
          if len(self.action_history) < 3:
         return False

          recent_actions = [h["action"].fingerprint for h in
          self.action_history[-5:]]
          if action.fingerprint in recent_actions:
         return True

          # Check semantic similarity for less obvious loops
          for recent in self.action_history[-3:]:
         similarity = self._compute_similarity(action, recent["action"])
         if similarity > self.similarity_threshold:
         return True

          return False
```

Bài học rút ra: Agent không tự nhiên nhận ra khi chúng bị kẹt. Bạn cần loop detection tường minh, và bạn cần giới hạn lượng “reflection” history giữ lại trong context.

Các Failure Mode Phổ Biến và Cách Nhận Biết

Sau hai năm chạy agent trong production, tôi đã phân loại các failure mode thường gặp nhất:

1. Resource Exhaustion (Cạn Kiệt Tài Nguyên)

Triệu chứng: Chi phí tăng đột biến, chạm rate limit, hệ thống chậm lại **Nguyên nhân gốc:** Không có budget, không có limit, không nhận thức về việc tiêu thụ tài nguyên **Phát hiện:** Monitor chi phí API, token usage, và thời gian thực thi theo thời gian thực

2. Goal Drift (Lệch Mục Tiêu)

Triệu chứng: Agent làm những việc đúng về mặt kỹ thuật nhưng lệch trọng tâm **Nguyên nhân gốc:** Hướng dẫn mơ hồ cho phép diễn giải sáng tạo **Phát hiện:** Con người thường xuyên sampling output của agent

3. Scope Creep (Phình Phạm Vi)

Triệu chứng: Agent “nhiệt tình” làm nhiều hơn yêu cầu **Nguyên nhân gốc:** Hướng dẫn như “cải thiện cái này” mà không có ranh giới rõ ràng **Phát hiện:** Audit log cho thấy các action ngoài pattern mong đợi

4. Context Collapse (Sụp Đổ Context)

Triệu chứng: Agent quên các hướng dẫn hoặc context trước đó **Nguyên nhân gốc:** Tác vụ chạy dài vượt quá context window **Phát hiện:** Agent hỏi những câu hỏi mà nó đã biết câu trả lời

5. Hallucinated Tools (Tool Ảo)

Triệu chứng: Agent cố sử dụng tool không tồn tại **Nguyên nhân gốc:** Training data của model bao gồm tool mà agent không có **Phát hiện:** Tool call fail với tên tool không xác định

Xây Dựng Kill Switch và Circuit Breaker

Mọi production agent đều cần nút dừng khẩn cấp. Đây là pattern tôi sử dụng:

```
class CircuitBreaker:
"""Production circuit breaker for AI agents."""

def __init__(self, agent_id: str, redis_client: Redis):
  self.agent_id = agent_id
  self.redis = redis_client
  self.failure_threshold = 5
  self.reset_timeout = 300  # 5 minutes

def check_kill_switch(self) -> bool:
  """Check if agent has been manually killed."""
  return self.redis.get(f"kill_switch:{self.agent_id}") == b"1"

def record_failure(self, error: str):
  """Record a failure and potentially trip the breaker."""
  key = f"failures:{self.agent_id}"
  pipe = self.redis.pipeline()
  pipe.rpush(key, f"{time.time()}:{error}")
  pipe.ltrim(key, -self.failure_threshold, -1)
  pipe.expire(key, self.reset_timeout)
  pipe.execute()

  failures = self.redis.llen(key)
  if failures >= self.failure_threshold:
 self._trip_breaker()

def _trip_breaker(self):
  """Automatically stop the agent."""
  self.redis.setex(
 f"breaker_tripped:{self.agent_id}",
 self.reset_timeout,
 "auto"
  )
  self._send_alert(f"Circuit breaker tripped for agent
  {self.agent_id}")

def can_proceed(self) -> bool:
  """Check all conditions before agent proceeds."""
  if self.check_kill_switch():
 raise AgentKilledException("Manual kill switch activated")

  if self.redis.get(f"breaker_tripped:{self.agent_id}"):
 raise CircuitBreakerException("Circuit breaker is tripped")

  return True
```

Kill switch được điều khiển qua một Redis key đơn giản có thể được set từ bất kỳ đâu—một Slack command, một monitoring alert, hoặc một phiên SSH hoảng loạn lúc 3 giờ sáng.

Sự thật phũ phàng: Mọi agent đều sẽ fail vào một lúc nào đó. Câu hỏi không phải là liệu bạn có cần kill switch và circuit breaker hay không—mà là liệu bạn sẽ triển khai chúng trước hay sau sự cố production đầu tiên.

Chương 5: Context Engineering

Nếu có một kỹ năng phân biệt agent hoạt động tốt với agent không hoạt động, đó là context engineering. Context window vừa là siêu năng lực vừa là gót chân Achilles của agent.

Context Window Không Phải Vô Hạn (Và Nó Đắt Đỏ)

Hãy làm một phép tính sẽ thay đổi cách bạn nghĩ về agent.

Claude 3.5 Sonnet có context window 200K token. Nghe có vẻ khổng lồ cho đến khi bạn nhận ra: - Một file codebase điển hình là 500-2000 token - Một stack trace với context xung quanh là 1000-3000 token - System prompt thường là 2000-5000 token - Mỗi tool call và response thêm 200-500 token

Trong một phiên debug, tôi đã chứng kiến context đầy lên như thế này:

```
System prompt ban đầu:         3,000 token
Tin nhắn đầu tiên của user:      500 token
Agent reasoning:               2,000 token
Tool: read_file (config.py):   1,200 token
Tool: read_file (main.py):     2,800 token
Tool: run_command (logs):      8,000 token
Agent phân tích:               1,500 token
Tool: read_file (utils.py):    1,100 token
Agent reasoning:               2,200 token
...

Sau 15 phút: đã dùng 180,000 token
```

Và đây là điều quan trọng: bạn trả tiền cho mọi token trong context window đó ở mỗi API call. Nếu context của bạn là 100K token và bạn thực hiện 20 API call trong một phiên, bạn đã trả cho 2 triệu input token.

Với giá Claude hiện tại ($3/triệu input token), đó là $6 chỉ riêng cho context. Cộng thêm output token và bạn đang nhìn vào $10-15 cho một phiên debug đơn lẻ.

Token Economics: Đo Lường và Tối Ưu Hóa Usage

Tôi theo dõi các metric này cho mọi agent:

```
        class TokenMetrics:
        def __init__(self):
          self.metrics = {
         "total_input_tokens": 0,
         "total_output_tokens": 0,
         "context_size_by_call": [],
         "tokens_per_task": [],
         "tool_call_token_overhead": []
          }

        def record_call(self, input_tokens: int, output_tokens: int,
```

```
     context_size: int, tool_calls: int):
  self.metrics["total_input_tokens"] += input_tokens
  self.metrics["total_output_tokens"] += output_tokens
  self.metrics["context_size_by_call"].append(context_size)
  self.metrics["tool_call_token_overhead"].append(
 tool_calls * 350  # Average overhead per tool call
  )

def get_cost_breakdown(self) -> dict:
  input_cost = self.metrics["total_input_tokens"] / 1_000_000 *
  3.0
  output_cost = self.metrics["total_output_tokens"] / 1_000_000
  * 15.0

  return {
 "input_cost_usd": input_cost,
 "output_cost_usd": output_cost,
 "total_cost_usd": input_cost + output_cost,
 "avg_context_size":
  statistics.mean(self.metrics["context_size_by_call"]),
 "context_growth_rate": self._calculate_growth_rate()
  }
```

Insight đã thay đổi cách tiếp cận của tôi: **tốc độ tăng trưởng context quan trọng hơn kích thước tuyệt đối**. Một agent tăng context 5K token mỗi lượt sẽ chạm limit nhanh hơn 4 lần so với agent tăng 1.25K.

Các Kỹ Thuật Nén Context

Khi context trở nên quá lớn, bạn có các lựa chọn:

1. Summarization (Tóm Tắt)

Thay thế log chi tiết bằng bản tóm tắt:

```
def compress_log_output(log_text: str, llm: LLM) -> str:
"""Compress verbose logs to key findings."""
if len(log_text) < 2000:
  return log_text

summary = llm.complete(f"""
Summarize these logs, keeping only:
- Error messages and their line numbers
- Warning patterns
- Key timestamps
- Relevant configuration values

Logs:
{log_text[:8000]}  # Sample if very large

Summary (be concise):
""")

return f"[Compressed from {len(log_text)} chars]\n{summary}"
```

2. Sliding Window (Cửa Sổ Trượt)

Chỉ giữ context gần đây:

```
def sliding_window_context(messages: list, max_tokens: int) ->
  list:
"""Keep most recent messages that fit in window."""

# Always keep system message
system_msg = messages[0] if messages[0]["role"] == "system" else
  None

# Add messages from newest to oldest until we hit limit
kept = []
current_tokens = count_tokens(system_msg) if system_msg else 0

for msg in reversed(messages[1:]):
  msg_tokens = count_tokens(msg)
  if current_tokens + msg_tokens > max_tokens:
 break
  kept.insert(0, msg)
  current_tokens += msg_tokens

if system_msg:
  kept.insert(0, system_msg)

return kept
```

3. Hierarchical Context (Context Phân Cấp)

Duy trì bản tóm tắt ở các mức độ chi tiết khác nhau:

```
class HierarchicalContext:
def __init__(self):
  self.immediate = []    # 5 lượt cuối, đầy đủ chi tiết
  self.recent = []       # 20 lượt cuối, đã tóm tắt
  self.session = ""      # Toàn bộ phiên, nén cao

def add_turn(self, turn: Turn):
  self.immediate.append(turn)

  # Rotate to recent when immediate fills up
  if len(self.immediate) > 5:
 old = self.immediate.pop(0)
 self.recent.append(self._summarize_turn(old))

  # Compress recent into session summary periodically
  if len(self.recent) > 20:
 self.session = self._compress_session()
 self.recent = self.recent[-5:]

def get_context(self) -> str:
  return f"""
Session overview: {self.session}

Recent activity: {self._format_recent()}

Current focus:
{self._format_immediate()}
"""
```

Khi Nào Nên Tóm Tắt vs Khi Nào Nên Quên

Cây quyết định này đã phục vụ tôi tốt:

```
Thông tin có tiềm năng cần thiết cho tác vụ hiện tại không?
├─ Không → Quên đi (thậm chí không cần tóm tắt)
└─ Có → Có khả năng cần nguyên văn không?
   ├─ Có → Giữ nguyên (code, config, error chính xác)
   └─ Không → Có chứa fact hoặc quyết định quan trọng không?
  ├─ Có → Tóm tắt và giữ
  └─ Không → Quên đi
```

Trong thực tế: - **Luôn giữ:** Thông báo lỗi, đường dẫn file được đề cập, preference của user đã nêu - **Tóm tắt:** Output của command, kết quả phân tích log, phát hiện khi khám phá - **Quên:** Log dài dòng đã phân tích, các hướng tiếp cận đã bỏ, xác nhận trùng lặp

Chuyện Thực Tế: Agent Quên Mất Nó Đang Làm Gì

Chúng tôi có một agent thực hiện database migration phức tạp. Nhiệm vụ bao gồm: 1. Phân tích schema hiện tại 2. Tạo migration script 3. Test trong staging 4. Áp dụng vào production với khả năng rollback

Agent làm tốt các bước 1-3. Nhưng đến bước 4, context đã đầy phân tích schema và kết quả test đến mức các yêu cầu migration ban đầu đã bị cuộn ra khỏi context window.

Agent áp dụng một migration đúng về mặt kỹ thuật dựa trên context gần đây nhưng bỏ lỡ một ràng buộc quan trọng từ yêu cầu ban đầu: duy trì backward compatibility với phiên bản API cũ.

Chúng tôi phát hiện trong staging (may quá), nhưng chỉ vì tình cờ có người test endpoint API cũ.

Cách sửa là triển khai cái tôi gọi là “pinned context”:

```
class PinnedContext:
"""Context items that should never be evicted."""

def __init__(self):
  self.pinned_items = []
  self.max_pinned_tokens = 10000

def pin(self, content: str, label: str, priority: int = 5):
  """Pin important context that must survive compression."""
  tokens = count_tokens(content)

  item = {
 "content": content,
 "label": label,
 "priority": priority,
 "tokens": tokens,
 "pinned_at": time.time()
  }

  self.pinned_items.append(item)
  self._enforce_limit()

def _enforce_limit(self):
  """Evict lowest priority pins if over limit."""
  total = sum(item["tokens"] for item in self.pinned_items)

  if total > self.max_pinned_tokens:
 # Sort by priority, then by age
 self.pinned_items.sort(key=lambda x: (x["priority"],
  -x["pinned_at"]))

 while total > self.max_pinned_tokens and self.pinned_items:
```

```
 removed = self.pinned_items.pop(0)
 total -= removed["tokens"]

def get_pinned_context(self) -> str:
  """Get all pinned items formatted for inclusion in prompt."""
  return "\n\n".join([
 f"[PINNED: {item['label']}]\n{item['content']}"
 for item in sorted(self.pinned_items, key=lambda x:
  -x["priority"])
  ])
```

Giờ đây, khi bắt đầu một tác vụ phức tạp, chúng tôi tường minh pin các yêu cầu quan trọng:

```
agent.context.pin(
content="""
MIGRATION REQUIREMENTS:
- Must maintain backward compatibility with API v1
- Zero downtime deployment
- Rollback capability for 24 hours
""",
label="Core Requirements",
priority=10  # Maximum priority, never evict
)
```

Bài học rút ra: Context của agent là bộ nhớ làm việc của nó. Thông tin quan trọng cần được bảo vệ tường minh, giống như file quan trọng cần được backup.

Chương 6: Hệ Thống Memory Thực Sự Hoạt Động

Chat history là một ảo tưởng. Nó có cảm giác như memory, nhưng thực ra chỉ là một buffer cuộn mà trở nên đắt đỏ nhanh chóng và cuối cùng sẽ tràn. Memory thực sự—loại cho phép agent thực sự hữu ích theo thời gian—đòi hỏi engineering có chủ đích.

Tại Sao Chat History Không Đủ

Xem xét điều gì xảy ra với chat history thuần túy:

Ngày 1: Bạn nói với agent về phong cách code ưa thích, lịch deployment, và service nào là critical.

Ngày 2: Phiên mới. Agent không biết gì. Bạn lặp lại.

Ngày 30: Bạn đã lặp lại preference hàng chục lần. Mỗi lần lặp lại tốn token. Bạn đang trả tiền để đào tạo lại agent của mình mỗi ngày.

Điều này không chỉ gây khó chịu—nó đắt đỏ và ngăn cản agent phát triển chuyên môn thực sự trong môi trường của bạn.

Thiết Kế Persistent Memory: File, Database, Vector Store

Tôi đã thử cả ba, và mỗi cái đều có chỗ đứng:

File-Based Memory

Tốt nhất cho: Cấu hình, preference, knowledge base nhỏ

```
class FileMemory:
def __init__(self, memory_dir: str = "~/.agent/memory"):
  self.memory_dir = os.path.expanduser(memory_dir)
  os.makedirs(self.memory_dir, exist_ok=True)

def remember(self, category: str, key: str, value: Any):
  """Store a memory item."""
  category_dir = os.path.join(self.memory_dir, category)
  os.makedirs(category_dir, exist_ok=True)

  filepath = os.path.join(category_dir, f"{key}.json")
  with open(filepath, 'w') as f:
 json.dump({
 "value": value,
 "stored_at": datetime.now().isoformat(),
 "access_count": 0
 }, f, indent=2)

def recall(self, category: str, key: str) -> Optional[Any]:
  """Retrieve a memory item."""
```

```
  filepath = os.path.join(self.memory_dir, category,
  f"{key}.json")

  if not os.path.exists(filepath):
 return None

  with open(filepath, 'r') as f:
 data = json.load(f)

  # Track access for relevance scoring
  data["access_count"] += 1
  data["last_accessed"] = datetime.now().isoformat()

  with open(filepath, 'w') as f:
 json.dump(data, f, indent=2)

  return data["value"]
```

Database Memory

Tốt nhất cho: Dữ liệu có cấu trúc, quan hệ, truy vấn

```
class DatabaseMemory:
def __init__(self, db_path: str = "~/.agent/memory.db"):
  self.conn = sqlite3.connect(os.path.expanduser(db_path))
  self._init_schema()

def _init_schema(self):
  self.conn.executescript("""
 CREATE TABLE IF NOT EXISTS memories (
 id INTEGER PRIMARY KEY,
 category TEXT NOT NULL,
 content TEXT NOT NULL,
 embedding BLOB,
 importance INTEGER DEFAULT 5,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
 accessed_at TIMESTAMP,
 access_count INTEGER DEFAULT 0
 );

 CREATE INDEX IF NOT EXISTS idx_category ON memories(category);
 CREATE INDEX IF NOT EXISTS idx_importance ON
  memories(importance);
  """)

def remember(self, content: str, category: str, importance: int
  = 5):
  self.conn.execute(
 "INSERT INTO memories (category, content, importance) VALUES
  (?, ?, ?)",
 (category, content, importance)
  )
  self.conn.commit()

def recall_recent(self, category: str = None, limit: int = 10)
  -> list:
  query = "SELECT content, importance FROM memories"
  params = []

  if category:
 query += " WHERE category = ?"
 params.append(category)

  query += " ORDER BY importance DESC, created_at DESC LIMIT ?"
  params.append(limit)
```

```
  return self.conn.execute(query, params).fetchall()
```

Vector Store Memory

Tốt nhất cho: Semantic search, tìm context liên quan

```
class VectorMemory:
def __init__(self, collection_name: str = "agent_memory"):
  import chromadb
  self.client = chromadb.PersistentClient(
  path="~/.agent/vectors")
  self.collection = self.client.get_or_create_collection(
  collection_name)

def remember(self, content: str, metadata: dict = None):
  """Store with automatic embedding."""
  doc_id = hashlib.md5(content.encode()).hexdigest()

  self.collection.upsert(
 documents=[content],
 metadatas=[metadata or {}],
 ids=[doc_id]
  )

def recall_similar(self, query: str, n_results: int = 5) -> list:
  """Find semantically similar memories."""
  results = self.collection.query(
 query_texts=[query],
 n_results=n_results
  )

  return [
 {"content": doc, "metadata": meta}
 for doc, meta in zip(results["documents"][0],
  results["metadatas"][0])
  ]
```

Pattern MEMORY.md: Đơn Giản Nhưng Hiệu Quả

Trước khi xây dựng hệ thống phức tạp, hãy xem xét file markdown khiêm tốn:

```
# Agent Memory

## User Preferences
- Thích response ngắn gọn hơn giải thích dài dòng
- Dùng vim keybinding, đề cập shortcut khi liên quan
- Timezone: Asia/Saigon (GMT+7)
- Deployment window: Thứ Ba-Thứ Năm, 10 AM - 4 PM

## Environment
- Primary language: Python 3.11
- Infrastructure: Kubernetes on AWS (EKS)
- CI/CD: GitHub Actions
- Secrets management: HashiCorp Vault

## Learned Patterns
### 2024-01-15: Database connections
Service của user dùng connection pool với max 20 connection.
Vấn đề performance thường trace về pool exhaustion.

### 2024-01-20: Deployment rollbacks
Rollback cần approval từ kênh Slack #ops-leads.
```

```
User có quyền approve cho staging only.

## Mistakes to Avoid
- Đừng suggest `rm` mà không xác nhận; dùng `trash` thay thế
- Đừng assume master branch; org này dùng 'main'
- Đừng tạo file trong /tmp; dùng ~/scratch thay thế
```

Agent đọc file này khi bắt đầu phiên và có thể cập nhật khi học được điều mới:

```
def update_memory_file(memory_path: str, section: str, content:
  str):
"""Append a new memory to the appropriate section."""

with open(memory_path, 'r') as f:
  current = f.read()

# Find section and append
section_pattern = f"## {section}\n"
if section_pattern in current:
  parts = current.split(section_pattern)
  # Find end of section (next ## or end of file)
  section_content = parts[1].split("\n## ")[0]
  new_section = section_content.rstrip() + f"\n\n###
  {datetime.now().strftime('%Y-%m-%d')}\n{content}\n"
  parts[1] = new_section + "\n## " + "\n##
  ".join(parts[1].split("\n## ")[1:])
  current = section_pattern.join(parts)

with open(memory_path, 'w') as f:
  f.write(current)
```

Pattern này đơn giản, con người đọc được, có thể version control, và hiệu quả đáng ngạc nhiên cho hầu hết use case.

Neural Memory và Semantic Search

Cho nhu cầu tinh vi hơn, kết hợp embedding với structured storage cho bạn tốt nhất của cả hai thế giới:

```
class NeuralMemory:
"""Memory system with semantic search and importance decay."""

def __init__(self):
  self.vector_store = VectorMemory()
  self.db = DatabaseMemory()

def remember(self, content: str, memory_type: str, importance:
  int = 5):
  """Store memory with both semantic and structured access."""

  # Store in database for structured queries
  memory_id = self.db.remember(content, memory_type, importance)

  # Store embedding for semantic search
  self.vector_store.remember(content, {
 "memory_id": memory_id,
 "type": memory_type,
 "importance": importance,
 "timestamp": datetime.now().isoformat()
  })

def recall(self, query: str, context: str = None) -> list:
```

```
      """Retrieve relevant memories using semantic search."""

      # Get semantically similar memories
      similar = self.vector_store.recall_similar(query, n_results=10)

      # Also get recent high-importance memories
      recent = self.db.recall_recent(limit=5)

      # Combine and deduplicate
      all_memories = self._merge_and_rank(similar, recent)

      return all_memories[:5]  # Return top 5

    def _merge_and_rank(self, similar: list, recent: list) -> list:
      """Rank memories by relevance and recency."""
      scored = []

      for mem in similar:
     score = mem.get("similarity", 0.5) *
      mem["metadata"].get("importance", 5)
     scored.append((score, mem["content"]))

      for content, importance in recent:
     # Boost recent memories
     scored.append((importance * 0.8, content))

      # Deduplicate and sort
      seen = set()
      unique = []
      for score, content in sorted(scored, reverse=True):
     content_hash = hash(content)
     if content_hash not in seen:
     seen.add(content_hash)
     unique.append(content)

      return unique
```

Chuyện Thực Tế: Xây Dựng Memory Sống Sót Qua Restart

Chúng tôi có một agent học được các quirk của infrastructure rất tuyệt vời—trong suốt một ngày. Nó biết service nào hay lỗi, deployment nào cần monitoring thêm, team member nào cần notify cho các system khác nhau.

Rồi server reboot.

Tất cả những gì học được? Mất sạch. Agent quay lại thành người mới bỡ ngỡ, hỏi về các service mà nó đã sửa hàng chục lần.

Vấn đề rất tinh vi: chúng tôi đang lưu memory, nhưng lưu trong session state, sống trong bộ nhớ. Khi process chết, memory cũng chết theo.

Cách sửa bao gồm ba phần:

1. Persistent storage từ đầu:

```
    class PersistentSession:
    def __init__(self, session_id: str):
      self.session_id = session_id
      self.storage_path = f"~/.agent/sessions/{session_id}"
      self._load_or_create()
```

```
def _load_or_create(self):
  path = os.path.expanduser(self.storage_path)
  if os.path.exists(path):
 with open(path, 'r') as f:
 self.state = json.load(f)
  else:
 self.state = {"memories": [], "context": {}, "created":
  datetime.now().isoformat()}

def _save(self):
  path = os.path.expanduser(self.storage_path)
  os.makedirs(os.path.dirname(path), exist_ok=True)
  with open(path, 'w') as f:
 json.dump(self.state, f, indent=2)

def remember(self, key: str, value: Any):
  self.state["memories"].append({
 "key": key,
 "value": value,
 "timestamp": datetime.now().isoformat()
  })
  self._save()  # Persist immediately
```

2. Memory consolidation khi shutdown:

```
def graceful_shutdown(session: PersistentSession, memory:
  NeuralMemory):
"""Consolidate session learnings into long-term memory."""

# Extract key learnings from session
session_summary = summarize_session(session.state)

# Store important learnings in long-term memory
for learning in session_summary.key_learnings:
  memory.remember(
 learning.content,
 memory_type="session_learning",
 importance=learning.importance
  )

# Mark session as cleanly closed
session.state["closed_at"] = datetime.now().isoformat()
session._save()
```

3. Memory reload khi startup:

```
def initialize_agent(agent: Agent, memory: NeuralMemory):
"""Bootstrap agent with relevant long-term memories."""

# Get recent learnings
recent = memory.recall("recent system learnings and user
  preferences")

# Add to agent's initial context
if recent:
  agent.system_prompt += f"""

## Relevant Memories from Previous Sessions:
{format_memories(recent)}

Use these memories to provide contextually appropriate
  assistance.
"""
```

Bài học rút ra: Memory không chỉ là về storage—nó là về toàn bộ lifecycle của remember, persist, recover, và apply. Bỏ lỡ bất kỳ bước nào và agent của bạn sẽ bị mất trí nhớ.

Chương 7: Điều Phối Đa Agent

Lần đầu tiên một trong những agent của tôi nói với tôi “Tôi cần giúp đỡ với phần này,” tôi nhận ra mình đã nghĩ sai về agent. Một agent đơn lẻ cố làm mọi thứ giống như một developer đơn lẻ cố trở thành chuyên gia frontend, DBA, chuyên gia bảo mật, và kỹ sư infrastructure cùng một lúc.

Khi Một Agent Không Đủ

Dấu hiệu bạn cần nhiều agent:

1. **Task đòi hỏi chuyên môn khác nhau:** Debug một hệ thống có thể cần một log analyst, một code reviewer, và một infrastructure specialist.
2. **Cơ hội song song hóa:** Kiểm tra 5 service có thể được thực hiện bởi 5 agent đồng thời.
3. **Tách biệt context:** Agent điều tra database không nên bị ô nhiễm với chi tiết networking trừ khi chúng liên quan.
4. **Cô lập rủi ro:** Agent có write access không nên là agent đang khám phá và thử nghiệm.

Pattern Orchestrator

Pattern multi-agent đáng tin cậy nhất tôi tìm thấy là orchestrator:

```
class Orchestrator:
"""Manager agent that delegates to specialist agents."""

def __init__(self, llm: LLM):
  self.llm = llm
  self.specialists = {
 "log_analyst": LogAnalystAgent(llm),
 "code_reviewer": CodeReviewAgent(llm),
 "infrastructure": InfrastructureAgent(llm),
 "security": SecurityAuditAgent(llm)
  }
  self.task_queue = []
  self.results = {}

def process_request(self, request: str) -> str:
  # First, understand and decompose the request
  plan = self._create_plan(request)

  # Execute plan steps, delegating to specialists
  for step in plan.steps:
 if step.requires_specialist:
 result = self._delegate(step)
 else:
 result = self._execute_directly(step)

 self.results[step.id] = result

  # Synthesize results into coherent response
  return self._synthesize(request, plan, self.results)
```

```
    def _create_plan(self, request: str) -> Plan:
      """Decompose request into steps with specialist assignments."""

      response = self.llm.complete(f"""
      Analyze this request and create an execution plan.

      Request: {request}

      Available specialists:
      - log_analyst: Analyzes logs, finds patterns, extracts
      relevant entries
      - code_reviewer: Reviews code, identifies issues, suggests
      fixes
      - infrastructure: Checks systems, networking, resources
      - security: Audits for vulnerabilities, checks configurations

      Create a plan with:
      1. Steps needed (can be parallel if independent)
      2. Which specialist handles each step (or "orchestrator" for
      simple tasks)
      3. Dependencies between steps

      Output as JSON.
      """)

      return Plan.from_json(response)

    def _delegate(self, step: PlanStep) -> StepResult:
      """Delegate a step to the appropriate specialist."""

      specialist = self.specialists.get(step.specialist)
      if not specialist:
     return StepResult(error=f"Unknown specialist:
      {step.specialist}")

      # Provide relevant context from previous steps
      context = self._gather_context(step.dependencies)

      return specialist.execute(step.task, context)
```

Giao Tiếp Giữa Các Agent: Shared Context, Message Passing

Agent cần chia sẻ thông tin, nhưng làm thế nào? Tôi đã thử ba cách tiếp cận:

1. Shared Memory Space

```
    class SharedMemory:
    """Thread-safe shared memory for multiple agents."""

    def __init__(self):
      self._data = {}
      self._lock = threading.Lock()

    def write(self, key: str, value: Any, agent_id: str):
      with self._lock:
     self._data[key] = {
     "value": value,
     "written_by": agent_id,
     "timestamp": time.time()
     }

    def read(self, key: str) -> Optional[Any]:
      with self._lock:
```

```
 entry = self._data.get(key)
 return entry["value"] if entry else None

def read_all_from(self, agent_id: str) -> dict:
  """Get all entries written by a specific agent."""
  with self._lock:
 return {
 k: v["value"]
 for k, v in self._data.items()
 if v["written_by"] == agent_id
 }
```

2. Message Passing

```
class AgentMessageBus:
"""Pub/sub message bus for agent communication."""

def __init__(self):
  self.queues = defaultdict(Queue)
  self.broadcasts = []

def send(self, from_agent: str, to_agent: str, message: dict):
  """Send a direct message to another agent."""
  self.queues[to_agent].put({
 "from": from_agent,
 "message": message,
 "timestamp": time.time()
  })

def broadcast(self, from_agent: str, message: dict):
  """Broadcast to all agents."""
  self.broadcasts.append({
 "from": from_agent,
 "message": message,
 "timestamp": time.time()
  })

def receive(self, agent_id: str, timeout: float = 0.1) ->
  Optional[dict]:
  """Check for messages (non-blocking by default)."""
  try:
 return self.queues[agent_id].get(timeout=timeout)
  except Empty:
 return None
```

3. Result Aggregation

```
class ResultAggregator:
"""Collect and synthesize results from multiple agents."""

def __init__(self, llm: LLM):
  self.llm = llm
  self.results = []

def add_result(self, agent_id: str, task: str, result: Any):
  self.results.append({
 "agent": agent_id,
 "task": task,
 "result": result,
 "timestamp": time.time()
  })

def synthesize(self, original_request: str) -> str:
  """Combine all results into a coherent response."""
```

```
          results_text = "\n\n".join([
         f"## {r['agent']}: {r['task']}\n{r['result']}"
         for r in self.results
          ])

          return self.llm.complete(f"""
          Original request: {original_request}

          Results from specialist agents:
          {results_text}

          Synthesize these findings into a coherent response that:
          1. Addresses the original request directly
          2. Highlights key findings from each specialist
          3. Notes any conflicts or uncertainties
          4. Provides clear next steps or recommendations
          """)
```

Thực Thi Agent Song Song vs Tuần Tự

Lựa chọn giữa thực thi song song và tuần tự phụ thuộc vào dependency:

```
        class AgentExecutor:
        """Execute agent tasks with proper dependency handling."""

        def __init__(self, max_parallel: int = 4):
          self.max_parallel = max_parallel
          self.executor = ThreadPoolExecutor(max_workers=max_parallel)

        def execute_plan(self, plan: Plan, agents: dict) -> dict:
          """Execute plan steps respecting dependencies."""

          results = {}
          completed = set()

          while len(completed) < len(plan.steps):
         # Find steps ready to execute (dependencies satisfied)
         ready = [
         step for step in plan.steps
         if step.id not in completed
         and all(dep in completed for dep in step.dependencies)
         ]

         if not ready:
         raise RuntimeError("Circular dependency detected")

         # Execute ready steps in parallel
         futures = {}
         for step in ready[:self.max_parallel]:
         agent = agents[step.specialist]
         context = {dep: results[dep] for dep in step.dependencies}
         futures[step.id] = self.executor.submit(
             agent.execute, step.task, context
         )

         # Collect results
         for step_id, future in futures.items():
         results[step_id] = future.result()
         completed.add(step_id)

          return results
```

Khi nào nên song song hóa: - Thu thập thông tin độc lập (kiểm tra nhiều service) - Phân tích nhiều file (mỗi file có thể phân tích riêng) - Cross-validation (có nhiều agent xác minh phát hiện)

Khi nào nên tuần tự: - Mỗi bước phụ thuộc vào kết quả trước - Ràng buộc tài nguyên (API rate limit, token budget) - Quản lý rủi ro (muốn human review giữa các bước)

Chuyện Thực Tế: Điều Phối Đội 4 Agent Cho Một Task Phức Tạp

Task có vẻ hợp lý: điều tra tại sao API response time của chúng tôi giảm trong tuần qua.

Tôi thiết lập bốn agent: - **LogAnalyst:** Parse application log tìm slow request - **MetricsAgent:** Phân tích Prometheus metric tìm pattern - **CodeReviewer:** Kiểm tra commit gần đây tìm performance issue - **InfraAgent:** Xem xét thay đổi infrastructure và resource usage

Orchestrator khởi động chúng song song. Trong vài giây, tôi có vấn đề.

Cả bốn agent đều đang gọi cùng một LLM API. Chúng tôi vượt rate limit. Request bắt đầu fail. Các agent, thấy failure, bắt đầu retry. Thêm nhiều rate limit hit. Orchestrator, không nhận được kết quả, bắt đầu spawn thêm request để kiểm tra các agent của nó.

Trong vòng 2 phút, chúng tôi đã thực hiện 400 API call và chỉ nhận được khoảng 50 response hữu ích.

Fix #1: Shared rate limiter

```
class SharedRateLimiter:
"""Rate limiter shared across all agents."""

def __init__(self, requests_per_minute: int = 50):
  self.requests_per_minute = requests_per_minute
  self.request_times = []
  self._lock = threading.Lock()

def acquire(self, timeout: float = 30) -> bool:
  """Wait for rate limit capacity."""
  deadline = time.time() + timeout

  while time.time() < deadline:
 with self._lock:
 now = time.time()
 # Remove requests older than 1 minute
 self.request_times = [t for t in self.request_times if now - t
  < 60]

 if len(self.request_times) < self.requests_per_minute:
     self.request_times.append(now)
     return True

 time.sleep(0.5)

  return False
```

Fix #2: Agent coordination protocol

```
class CoordinatedAgent:
"""Agent that coordinates with others before acting."""

def __init__(self, agent_id: str, coordinator: AgentCoordinator):
  self.agent_id = agent_id
```

```
  self.coordinator = coordinator

def execute(self, task: str, context: dict) -> Any:
  # Announce what we're about to do
  self.coordinator.announce_intent(self.agent_id, task)

  # Check if another agent is already handling something similar
  similar = self.coordinator.find_similar_intents(task)
  if similar:
 # Wait for their result instead of duplicating work
 return self.coordinator.wait_for_result(similar[0])

  # Acquire rate limit slot
  if not self.coordinator.rate_limiter.acquire():
 return AgentResult(error="Rate limit timeout")

  # Do the work
  result = self._do_work(task, context)

  # Share result with other agents
  self.coordinator.publish_result(self.agent_id, task, result)

  return result
```

Fix #3: Staggered startup

```
def launch_agents_staggered(agents: list, delay_seconds: float =
  2):
"""Launch agents with staggered timing to prevent thundering
  herd."""

futures = []
for i, agent in enumerate(agents):
  # Wait before launching each agent
  time.sleep(delay_seconds)
  future = executor.submit(agent.start)
  futures.append(future)

return futures
```

Sau các fix này, cuộc điều tra chạy mượt mà. Log analyst tìm thấy slow database query. Metrics agent xác nhận query latency tăng. Code reviewer tìm thấy thay đổi gần đây đã xóa database index. Infrastructure agent xác minh database resource usage bình thường.

Orchestrator tổng hợp: “Response time degradation gây ra bởi missing database index sau commit abc123 vào thứ Ba. Khuyến nghị: khôi phục index với migration script.”

Tổng thời gian: 4 phút. Tổng API call: 47 (trong limit). Vấn đề đã giải quyết.

Bài học rút ra: Hệ thống multi-agent nhân đôi cả khả năng lẫn failure mode. Coordination không phải tùy chọn—nó là thiết yếu. Mọi tài nguyên shared đều cần protection, mọi agent cần nhận thức về các agent khác, và orchestrator cần visibility vào toàn bộ trạng thái hệ thống.

Tóm Tắt Phần 2: Những Bài Học Đắt Giá

Bốn chương này chứa đựng những vết sẹo từ hành trình production của tôi với AI agent. Nếu tôi có thể đưa ra một takeaway từ mỗi chương:

Chương 4: Mọi agent đều cần kill switch và circuit breaker. Xây dựng chúng trước, không phải sau sự cố đầu tiên.

Chương 5: Context đắt đỏ và hữu hạn. Engineer nó có chủ đích—compress, summarize, pin, và evict với mục đích rõ ràng.

Chương 6: Memory không tự động. Xây persistence vào agent của bạn từ ngày đầu, hoặc chấp nhận rằng nó sẽ quên mọi thứ quan trọng.

Chương 7: Hệ thống multi-agent đòi hỏi infrastructure coordination. Shared resource, rate limit, và communication protocol quan trọng như chính các agent.

Những bài học đắt giá không phải là học tập tùy chọn—chúng là giá vé vào cổng của production AI agent. Trả học phí trước bằng cách học từ lỗi của người khác, hoặc trả sau với các sự cố của chính bạn.

Tiếp theo, trong Phần 3, chúng ta sẽ biến những bài học này thành production pattern: security, quản lý chi phí, reliability, và thiết kế human-in-the-loop vô cùng quan trọng.

PHẦN 3: CÁC MẪU THIẾT KẾ CHO PRODUCTION

Chương 8: Bảo mật và Kiểm soát Truy cập

Sức Mạnh Đáng Sợ của một Agent có Quyền SSH

Hãy để tôi vẽ cho bạn một bức tranh. 2 giờ sáng thứ Tư. Bạn vừa cấp quyền SSH cho AI agent mới toanh của mình để truy cập vào máy chủ production vì nó cần kiểm tra log và khởi động lại các service. Agent này rất thông minh—nó có thể chẩn đoán vấn đề, chạy các lệnh, và sửa lỗi nhanh hơn bất kỳ kỹ sư on-call nào.

Ba mươi phút sau, bạn đang nhìn chằm chằm vào terminal trong sự hoang mang. Agent, trong nỗ lực “dọn dẹp dung lượng ổ đĩa,” đã rm -rf một thư mục mà nó không nên chạm vào. Không phải cố ý—nó thực sự nghĩ rằng nó đang giúp đỡ. Model đã thấy các lệnh tương tự trong dữ liệu training để dọn dẹp các thư mục tạm. Thư mục của bạn tình cờ khớp với pattern đó.

Đây là thách thức cơ bản của bảo mật agent: bạn đang trao quyền ra quyết định tự động cho một hệ thống thiếu sự hiểu biết thực sự về hậu quả. Không giống như một kỹ sư junior có thể ngần ngại trước khi chạy một lệnh phá hoại, một agent sẽ thực thi với cùng một sự tự tin dù nó đang đọc file hay xóa database của bạn.

Nguyên tắc Đặc quyền Tối thiểu cho AI Agent

Nguyên tắc bảo mật mà các agent của bạn cần tuân theo không mới—đó là nguyên tắc tương tự mà chúng ta đã áp dụng cho service account trong nhiều thập kỷ: chỉ cấp những quyền cần thiết để hoàn thành nhiệm vụ, không hơn.

Nhưng đây là điểm khó với các agent. Một service account truyền thống chạy một tập hợp các thao tác đã biết, xác định. Một agent, theo định nghĩa, đưa ra quyết định tại runtime về các thao tác cần thực hiện. Làm sao bạn cấp đặc quyền tối thiểu cho thứ mà hành vi của nó không hoàn toàn dự đoán được?

Câu trả lời là các hạn chế theo lớp:

```yaml
# agent-permissions.yaml
agent:
  name: log-analyzer
  allowed_actions:
read:
 - /var/log/**
 - /app/logs/**
execute:
 - grep
 - tail
 - head
 - cat
```

```
 - less
 - jq
forbidden:
 - rm
 - mv
 - dd
 - chmod
 - chown
 - sudo
 - curl
 - wget

  resource_limits:
max_file_size_read: 100MB
max_command_duration: 60s
max_concurrent_commands: 3

  network:
allowed_hosts:
 - metrics.internal:9090
blocked_hosts:
 - "*"  # default deny
```

Cấu hình này tạo ra một hành lang hẹp của các hành vi được phép. Agent có thể đọc log, sử dụng các công cụ xử lý văn bản, và truy vấn các metric nội bộ—không gì khác. Ngay cả khi LLM bên dưới quyết định muốn tải xuống thứ gì đó từ internet hoặc sửa đổi file, tầng thực thi sẽ chặn nó.

Các Chiến lược Sandbox

Các chiến lược sandbox khác nhau cung cấp sự đánh đổi khác nhau giữa bảo mật và khả năng:

Cấp độ 1: Restricted Shell

Cách tiếp cận đơn giản nhất là môi trường shell bị hạn chế. Các công cụ như `rbash` hoặc các wrapper script tùy chỉnh giới hạn những lệnh nào khả dụng:

```
#!/bin/bash
# agent-shell.sh - A restricted shell for agent execution

ALLOWED_COMMANDS="ls cat grep tail head jq kubectl docker"

execute_command() {
local cmd="$1"
local base_cmd=$(echo "$cmd" | awk '{print $1}')

# Check if command is allowed
if ! echo "$ALLOWED_COMMANDS" | grep -qw "$base_cmd"; then
  echo "ERROR: Command '$base_cmd' is not allowed"
  exit 1
fi

# Check for dangerous patterns
if echo "$cmd" | grep -qE '(rm|>|>>|\|.*rm|;.*rm)'; then
  echo "ERROR: Potentially destructive pattern detected"
  exit 1
fi

# Execute with timeout
timeout 60s bash -c "$cmd"
}
```

Cách này nhanh và nhẹ nhưng cung cấp sự cô lập tối thiểu. Một kẻ tấn công quyết tâm (hoặc agent bị nhầm lẫn) có thể tìm cách vượt qua các hạn chế này.

Cấp độ 2: Cô lập bằng Container

Chạy các hành động của agent bên trong các container cung cấp sự cô lập mạnh hơn:

```python
# agent_sandbox.py
import docker
import tempfile
from typing import Optional

class AgentSandbox:
def __init__(self):
  self.client = docker.from_env()
  self.image = "agent-sandbox:latest"

def execute(
  self,
  command: str,
  timeout: int = 60,
  memory_limit: str = "256m",
  cpu_quota: int = 50000,  # 50% of one CPU
  network_mode: str = "none"
) -> dict:
  """Execute a command in an isolated container."""

  with tempfile.TemporaryDirectory() as tmpdir:
 container = self.client.containers.run(
 self.image,
 command=f"sh -c '{command}'",
 detach=True,
 mem_limit=memory_limit,
 cpu_quota=cpu_quota,
 network_mode=network_mode,
 read_only=True,
 volumes={
     tmpdir: {'bind': '/workspace', 'mode': 'rw'},
     '/var/log': {'bind': '/logs', 'mode': 'ro'},  # read-only
 },
 user="nobody",
 security_opt=["no-new-privileges:true"],
 )

 try:
 result = container.wait(timeout=timeout)
 logs = container.logs().decode('utf-8')
 return {
     "exit_code": result["StatusCode"],
     "output": logs,
     "error": None
 }
 except Exception as e:
 container.kill()
 return {
     "exit_code": -1,
     "output": "",
     "error": str(e)
 }
 finally:
 container.remove(force=True)
```

Dockerfile cho sandbox image nên tối giản:

```
FROM alpine:3.19

RUN apk add --no-cache \
bash \
coreutils \
grep \
jq \
curl \
&& rm -rf /var/cache/apk/*

# Create non-root user
RUN adduser -D -s /bin/bash agent

USER agent
WORKDIR /workspace
```

Cấp độ 3: Cô lập bằng VM

Đối với các yêu cầu bảo mật cao nhất—đặc biệt khi agent xử lý các thao tác nhạy cảm hoặc đầu vào không đáng tin cậy—cô lập bằng VM sử dụng Firecracker hoặc gVisor cung cấp sự cô lập gần cấp độ phần cứng:

```
# For extreme isolation, consider Firecracker microVMs
# This is typically overkill for most use cases, but critical
# when agents are exposed to untrusted inputs

class FirecrackerSandbox:
"""
Each agent action runs in a fresh microVM that boots in ~125ms.
Complete isolation from the host kernel.
"""

def execute(self, command: str) -> dict:
  # Firecracker implementation details omitted
  # See: https://firecracker-microvm.github.io/
  pass
```

Audit Logging: Biết Agent của Bạn Đã Làm Gì

Khi một agent hành động tự động, bạn cần ghi nhớ hoàn hảo mọi hành động nó thực hiện. Điều này không phải tùy chọn—đó là cách bạn debug lỗi, điều tra sự cố, và xây dựng niềm tin vào hệ thống agent của bạn.

Đây là mẫu audit logging tôi sử dụng trong production:

```
# audit_logger.py
import json
import hashlib
from datetime import datetime
from typing import Any, Optional
import structlog

class AgentAuditLogger:
def __init__(self, agent_id: str, session_id: str):
  self.agent_id = agent_id
  self.session_id = session_id
  self.sequence = 0
  self.logger = structlog.get_logger()

def log_action(
  self,
```

```
          action_type: str,
          action_input: Any,
          action_output: Any,
          success: bool,
          duration_ms: float,
          context: Optional[dict] = None
        ):
          """Log every agent action with full context."""

          self.sequence += 1

          # Create deterministic action ID
          action_id = hashlib.sha256(
         f"{self.session_id}:{self.sequence}".encode()
          ).hexdigest()[:12]

          log_entry = {
         "timestamp": datetime.utcnow().isoformat(),
         "agent_id": self.agent_id,
         "session_id": self.session_id,
         "action_id": action_id,
         "sequence": self.sequence,
         "action_type": action_type,
         "input": self._sanitize(action_input),
         "output": self._truncate(action_output),
         "success": success,
         "duration_ms": duration_ms,
         "context": context or {}
          }

          # Log to structured logging system
          self.logger.info(
         "agent_action",
         **log_entry
          )

          # Also write to append-only audit file
          with open(f"/var/log/agent-audit/{self.session_id}.jsonl",
          "a") as f:
         f.write(json.dumps(log_entry) + "\n")

        def _sanitize(self, data: Any) -> Any:
          """Remove sensitive data before logging."""
          if isinstance(data, str):
         # Redact common secret patterns
         patterns = [
         (r'password["\s:=]+["\']?[\w\-]+["\']?', 'password=<REDACTED>'),
         (r'api[_-]?key["\s:=]+["\']?[\w\-]+["\']?',
          'api_key=<REDACTED>'),
         (r'bearer\s+[\w\-\.]+', 'Bearer <REDACTED>'),
         ]
         result = data
         for pattern, replacement in patterns:
         result = re.sub(pattern, replacement, result,
          flags=re.IGNORECASE)
         return result
          return data

        def _truncate(self, data: Any, max_len: int = 10000) -> Any:
          """Truncate large outputs to prevent log bloat."""
          if isinstance(data, str) and len(data) > max_len:
         return data[:max_len] + f"... [truncated, total length:
          {len(data)}]"
          return data
```

Cách sử dụng trong vòng lặp thực thi agent của bạn:

```
audit = AgentAuditLogger(agent_id="infra-bot",
  session_id=session_id)

for action in agent.plan():
start = time.time()
try:
  result = sandbox.execute(action.command)
  audit.log_action(
 action_type=action.type,
 action_input=action.command,
 action_output=result["output"],
 success=result["exit_code"] == 0,
 duration_ms=(time.time() - start) * 1000,
 context={"reasoning": action.reasoning}
  )
except Exception as e:
  audit.log_action(
 action_type=action.type,
 action_input=action.command,
 action_output=None,
 success=False,
 duration_ms=(time.time() - start) * 1000,
 context={"error": str(e)}
  )
  raise
```

Câu chuyện Thực chiến: Agent Suýt Lộ Secret

Hãy để tôi kể cho bạn về lần suýt gặp nguy hiểm nhất với bảo mật agent.

Chúng tôi có một agent được thiết kế để giúp debug các vấn đề ứng dụng. Nó có thể đọc log, kiểm tra metric, và xem các file cấu hình. Những thứ tiêu chuẩn. Chúng tôi đã thiết lập sandbox khá tốt—cô lập container, mount read-only, không có quyền truy cập mạng.

Một ngày, một developer yêu cầu agent “tìm tại sao payment service đang fail.” Agent đã làm công việc của nó một cách xuất sắc. Nó tìm thấy lỗi trong log, truy ngược đến vấn đề cấu hình, và trình bày một phân tích chi tiết.

Nhưng đây là điều chúng tôi bỏ lỡ: file cấu hình mà nó đọc chứa database credential. Và agent, vì muốn giúp đỡ, đã bao gồm đoạn config liên quan trong phản hồi của nó—cả credential nữa.

Phản hồi được hiển thị trong một kênh Slack. Ba mươi kỹ sư nhìn thấy nó trước khi ai đó chú ý và xóa tin nhắn. Những credential đó phải được rotate ngay lập tức. Thật là một mớ hỗn độn.

Việc sửa chữa đòi hỏi nhiều thay đổi:

```
# secret_filter.py
import re
from typing import List, Tuple

class SecretFilter:
"""Filter secrets from agent outputs before displaying to
  users."""

PATTERNS: List[Tuple[str, str]] = [
  # Database URLs
  (r'(postgres|mysql|mongodb)://[^:]+:[^@]+@',
  r'\1://****:****@'),
  # API Keys (common formats)
  (r'(["\']?api[_-]?key["\']?\s*[:=]\s*["\']?)([\w\-]{20,})',
  r'\1****'),
```

```
  # AWS Keys
  (r'(AKIA[A-Z0-9]{16})', '****AWS_KEY****'),
  (r'(["\']?aws[_-]?secret["\']?\s*[:=]\s*["\']?)([\w/+=]{40})',
  r'\1****'),
  # Generic passwords
  (r'(["\']?password["\']?\s*[:=]\s*["\']?)([^\s"\',]+)',
  r'\1****'),
  # Bearer tokens
  (r'(Bearer\s+)[\w\-\.]+', r'\1****'),
  # Private keys
  (r'-----BEGIN [A-Z]+ PRIVATE KEY-----.*?-----END [A-Z]+
  PRIVATE KEY-----',
   '****PRIVATE_KEY****'),
]

def filter(self, text: str) -> str:
  result = text
  for pattern, replacement in self.PATTERNS:
 result = re.sub(pattern, replacement, result,
  flags=re.IGNORECASE | re.DOTALL)
  return result

def contains_secrets(self, text: str) -> bool:
  """Check if text likely contains secrets."""
  filtered = self.filter(text)
  return filtered != text

# Use in your agent response handler
filter = SecretFilter()

def present_agent_response(response: str) -> str:
filtered = filter.filter(response)
if filter.contains_secrets(response):
  # Log that secrets were filtered
  audit.log_action(
 action_type="secret_filter",
 action_input="[redacted]",
 action_output="secrets detected and filtered",
 success=True,
 duration_ms=0
  )
return filtered
```

Nhưng lọc thôi là chưa đủ. Giải pháp thực sự là ngăn agent truy cập secret ngay từ đầu:

```
# Mount configs without secrets
volumes:
  # Don't mount the real config
  # - /app/config:/config:ro

  # Mount a sanitized version
  - /app/config-sanitized:/config:ro

# Use a sidecar to generate sanitized configs
# that replace secrets with placeholders
```

Bài học: giả định rằng agent của bạn sẽ cố gắng bao gồm mọi thứ nó đọc trong output. Lọc tại nguồn, không chỉ tại thời điểm hiển thị.

Chương 9: Quản lý Chi phí

Chi phí Token Tăng Nhanh: Số liệu Thực từ Production

Hãy để tôi chia sẻ một số số liệu thực từ việc chạy AI agent trong production. Những con số này sẽ khiến bạn gật đầu đồng tình hoặc há hốc mồm kinh ngạc, tùy thuộc vào việc bạn đã làm điều này trước đó hay chưa.

Một phiên debug phức tạp đơn lẻ—nơi agent đọc log, phân tích pattern, kiểm tra nhiều service, và đưa ra khuyến nghị—có thể dễ dàng tiêu thụ 50.000-100.000 token. Với các model cấp GPT-4, đó là $1.50-$3.00 cho một phiên.

Nghe có vẻ quản lý được? Bây giờ hãy nhân với mức sử dụng:

- 10 kỹ sư sử dụng agent 5 lần mỗi ngày
- 22 ngày làm việc mỗi tháng
- Trung bình 75.000 token mỗi phiên

Đó là 10 × 5 × 22 × 75.000 = 82,5 triệu token mỗi tháng, hoặc khoảng $2.500/tháng chỉ cho một use case.

Và đây là điều bắt những team khác ngờ nhất: agent loop. Khi một agent rơi vào vòng lặp retry hoặc bị kẹt cố gắng giải quyết một vấn đề không thể giải quyết, mức sử dụng token bùng nổ. Tôi đã thấy các phiên đơn lẻ đạt 500.000 token trước khi bị kill—$15 cho một task thất bại.

Các Chiến lược Caching: Đừng Hỏi Cùng Một Câu Hỏi Hai Lần

Tối ưu chi phí có tác động lớn nhất rất đơn giản: đừng gọi LLM nếu bạn không cần.

Response Caching

Với các truy vấn deterministic (cùng context, cùng câu hỏi = cùng câu trả lời), cache response:

```
# llm_cache.py
import hashlib
import json
import redis
from typing import Optional, Any

class LLMCache:
def __init__(self, redis_url: str, default_ttl: int = 3600):
  self.redis = redis.from_url(redis_url)
  self.default_ttl = default_ttl

def _cache_key(self, prompt: str, model: str, params: dict) ->
  str:
  """Generate deterministic cache key."""
  content = json.dumps({
 "prompt": prompt,
 "model": model,
 "params": params
  }, sort_keys=True)
  return f"llm:v1:{hashlib.sha256(content.encode()).hexdigest()}"

def get(self, prompt: str, model: str, params: dict) ->
  Optional[str]:
```

```
  key = self._cache_key(prompt, model, params)
  cached = self.redis.get(key)
  if cached:
 return json.loads(cached)["response"]
  return None

def set(
  self,
  prompt: str,
  model: str,
  params: dict,
  response: str,
  ttl: Optional[int] = None
):
  key = self._cache_key(prompt, model, params)
  self.redis.setex(
 key,
 ttl or self.default_ttl,
 json.dumps({"response": response})
  )

# Usage
cache = LLMCache("redis://localhost:6379")

def call_llm(prompt: str, model: str, **params) -> str:
# Check cache first
cached = cache.get(prompt, model, params)
if cached:
  metrics.increment("llm_cache_hit")
  return cached

# Cache miss - call the API
metrics.increment("llm_cache_miss")
response = openai.chat.completions.create(
  model=model,
  messages=[{"role": "user", "content": prompt}],
  **params
)
result = response.choices[0].message.content

# Cache the response
cache.set(prompt, model, params, result)
return result
```

Semantic Caching

Với các truy vấn tương tự nhưng không giống hệt, sử dụng semantic caching dựa trên embedding:

```
# semantic_cache.py
import numpy as np
from typing import Optional, Tuple

class SemanticCache:
def __init__(
  self,
  embedding_model: str = "text-embedding-3-small",
  similarity_threshold: float = 0.95
):
  self.embedding_model = embedding_model
  self.threshold = similarity_threshold
  self.cache: list[Tuple[np.ndarray, str, str]] = []  #
  (embedding, prompt, response)

def _embed(self, text: str) -> np.ndarray:
  response = openai.embeddings.create(
 model=self.embedding_model,
```

```
 input=text
  )
  return np.array(response.data[0].embedding)

def _cosine_similarity(self, a: np.ndarray, b: np.ndarray) ->
  float:
  return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))

def get(self, prompt: str) -> Optional[str]:
  if not self.cache:
 return None

  query_embedding = self._embed(prompt)

  best_similarity = 0
  best_response = None

  for cached_embedding, cached_prompt, cached_response in
  self.cache:
 similarity = self._cosine_similarity(query_embedding,
  cached_embedding)
 if similarity > best_similarity:
 best_similarity = similarity
 best_response = cached_response

  if best_similarity >= self.threshold:
 return best_response
  return None

def set(self, prompt: str, response: str):
  embedding = self._embed(prompt)
  self.cache.append((embedding, prompt, response))
```

Lựa chọn Model: Khi nào Dùng Model Đắt vs Rẻ

Không phải mọi task đều cần GPT-4. Đây là chiến lược phân tầng model mà tôi sử dụng:

```
# model_router.py
from enum import Enum
from typing import Optional

class TaskComplexity(Enum):
SIMPLE = "simple"      # Classification, extraction, simple Q&A
MODERATE = "moderate"  # Analysis, summarization, code review
COMPLEX = "complex"    # Multi-step reasoning, complex code gen
CRITICAL = "critical"  # Production changes, security decisions

class ModelRouter:
MODEL_MAP = {
  TaskComplexity.SIMPLE: "gpt-3.5-turbo",
  TaskComplexity.MODERATE: "gpt-4o-mini",
  TaskComplexity.COMPLEX: "gpt-4o",
  TaskComplexity.CRITICAL: "gpt-4o",  # Plus human review
}

COST_PER_1K_TOKENS = {
  "gpt-3.5-turbo": 0.0015,
  "gpt-4o-mini": 0.00015,
  "gpt-4o": 0.005,
}

def route(
  self,
  task_type: str,
  context_size: int,
  requires_tools: bool = False
) -> str:
```

```
  """Select appropriate model based on task characteristics."""

  # Classify task complexity
  if task_type in ["classify", "extract", "format"]:
 complexity = TaskComplexity.SIMPLE
  elif task_type in ["analyze", "summarize", "review"]:
 complexity = TaskComplexity.MODERATE
  elif task_type in ["plan", "debug", "generate"]:
 complexity = TaskComplexity.COMPLEX
  elif task_type in ["deploy", "security", "delete"]:
 complexity = TaskComplexity.CRITICAL
  else:
 complexity = TaskComplexity.MODERATE

  # Upgrade if context is large (needs better attention)
  if context_size > 50000 and complexity ==
  TaskComplexity.SIMPLE:
 complexity = TaskComplexity.MODERATE

  # Tool use works better with capable models
  if requires_tools and complexity == TaskComplexity.SIMPLE:
 complexity = TaskComplexity.MODERATE

  return self.MODEL_MAP[complexity]

# Example usage in agent
router = ModelRouter()

def agent_think(task: str, context: str, tools: list) -> str:
model = router.route(
  task_type=classify_task(task),
  context_size=len(context),
  requires_tools=len(tools) > 0
)

return call_llm(
  prompt=f"{context}\n\nTask: {task}",
  model=model,
  tools=tools
)
```

Rate Limiting và Quota

Bảo vệ bản thân khỏi chi phí vượt tầm kiểm soát với các giới hạn cứng:

```
# cost_limiter.py
import time
from dataclasses import dataclass
from typing import Optional
import redis

@dataclass
class CostLimits:
per_request_tokens: int = 50000
per_session_tokens: int = 200000
per_user_daily_tokens: int = 1000000
per_user_daily_dollars: float = 50.0
global_daily_dollars: float = 500.0

class CostLimiter:
def __init__(self, redis_url: str, limits: CostLimits):
  self.redis = redis.from_url(redis_url)
  self.limits = limits

def check_and_reserve(
  self,
  user_id: str,
```

```
      session_id: str,
      estimated_tokens: int
    ) -> Tuple[bool, Optional[str]]:
      """Check if request is within limits and reserve capacity."""

      today = time.strftime("%Y-%m-%d")

      # Check per-request limit
      if estimated_tokens > self.limits.per_request_tokens:
     return False, f"Request exceeds token limit ({estimated_tokens}
      > {self.limits.per_request_tokens})"

      # Check session total
      session_key = f"tokens:session:{session_id}"
      session_total = int(self.redis.get(session_key) or 0)
      if session_total + estimated_tokens >
      self.limits.per_session_tokens:
     return False, f"Session token limit exceeded"

      # Check user daily limit
      user_key = f"tokens:user:{user_id}:{today}"
      user_total = int(self.redis.get(user_key) or 0)
      if user_total + estimated_tokens >
      self.limits.per_user_daily_tokens:
     return False, f"Daily user token limit exceeded"

      # Check global daily limit
      global_key = f"cost:global:{today}"
      global_cost = float(self.redis.get(global_key) or 0)
      estimated_cost = estimated_tokens * 0.00003  # rough estimate
      if global_cost + estimated_cost >
      self.limits.global_daily_dollars:
     return False, f"Global daily cost limit exceeded"

      # Reserve capacity
      pipe = self.redis.pipeline()
      pipe.incrby(session_key, estimated_tokens)
      pipe.expire(session_key, 86400)
      pipe.incrby(user_key, estimated_tokens)
      pipe.expire(user_key, 86400)
      pipe.execute()

      return True, None

    def record_actual(
      self,
      user_id: str,
      session_id: str,
      actual_tokens: int,
      actual_cost: float
    ):
      """Record actual usage after request completes."""
      today = time.strftime("%Y-%m-%d")
      global_key = f"cost:global:{today}"

      self.redis.incrbyfloat(global_key, actual_cost)
      self.redis.expire(global_key, 86400)
```

Câu chuyện Thực chiến: Tháng $500 và Cách Tránh Nó

Tháng đầu tiên chạy AI agent trong production, chúng tôi đạt $500 chi phí API. Để so sánh, chúng tôi đã budget $100.

Đây là những gì đã xảy ra: chúng tôi có một agent giúp review code. Developer yêu thích nó—nó bắt bug, đề xuất cải thiện, và sẵn sàng 24/7. Mức sử dụng tăng tự nhiên.

Những gì chúng tôi không tính đến: 1. Các PR lớn với hàng nghìn dòng tạo ra context khổng lồ 2. Developer hỏi các câu hỏi tiếp theo, mỗi câu đều yêu cầu toàn bộ context 3. Một số developer dùng nó như một rubber duck, trò chuyện rất nhiều 4. Agent đôi khi “bị kẹt” phân tích code phức tạp, đốt token

Cách sửa chữa là đa chiều:

```
# The changes we implemented

# 1. Context windowing - don't send entire files
def smart_context(files: list[str], question: str) -> str:
"""Only include relevant portions of files."""
relevant_chunks = []
for file in files:
  chunks = chunk_file(file, chunk_size=500)
  # Use embeddings to find relevant chunks
  relevant = find_relevant_chunks(chunks, question, top_k=3)
  relevant_chunks.extend(relevant)
return "\n---\n".join(relevant_chunks)

# 2. Session limits - force new sessions for different topics
MAX_TOKENS_PER_SESSION = 100000
MAX_MESSAGES_PER_SESSION = 20

# 3. User quotas - visible in UI
def get_user_quota_status(user_id: str) -> dict:
"""Show users their remaining quota."""
used = get_user_daily_usage(user_id)
limit = get_user_daily_limit(user_id)
return {
  "used": used,
  "limit": limit,
  "remaining": limit - used,
  "percentage": (used / limit) * 100
}

# 4. Circuit breaker - kill stuck sessions
class SessionCircuitBreaker:
def __init__(self, max_tokens: int = 50000, max_duration: int =
  300):
  self.max_tokens = max_tokens
  self.max_duration = max_duration

def check(self, session: AgentSession) -> bool:
  if session.total_tokens > self.max_tokens:
 session.terminate("Token limit exceeded")
 return False
  if session.duration_seconds > self.max_duration:
 session.terminate("Session timeout")
 return False
  return True
```

Sau khi triển khai các biện pháp kiểm soát này, chi phí của chúng tôi ổn định ở khoảng $150/tháng—50% vượt budget, nhưng có thể quản lý được. Quan trọng hơn, chúng tôi có khả năng nhìn thấy và kiểm soát.

Insight quan trọng: **hãy đối xử với các API call LLM như các truy vấn database vào đầu những năm 2000.** Chúng đắt đỏ, chúng có thể chạy ngoài tầm kiểm soát, và bạn cần monitoring và giới hạn từ ngày đầu tiên.

Chương 10: Độ tin cậy và Quan sát

Agent Cũng Cần Monitoring

Đây là một sự thật mà tôi mất quá lâu để học: monitoring một AI agent không giống như monitoring một service truyền thống. Với một service bình thường, bạn quan tâm đến uptime, latency, và tỷ lệ lỗi. Với một agent, bạn cũng cần monitor *chất lượng*—agent có thực sự giúp ích không, hay nó tự tin sản xuất rác?

Ba trụ cột của observability cho agent: 1. **Metric vận hành**: Agent có đang chạy không? Nhanh như thế nào? Có lỗi gì không? 2. **Metric chi phí**: Sử dụng token, chi phí API, tỷ lệ cache hit 3. **Metric chất lượng**: Tỷ lệ thành công, sự hài lòng của người dùng, hoàn thành task

Structured Logging cho Các Hành động của Agent

Mỗi hành động của agent nên tạo ra các log có cấu trúc có thể được truy vấn, tổng hợp, và cảnh báo:

```
# agent_logging.py
import structlog
from contextvars import ContextVar
from typing import Any

# Context variables for request tracing
request_id_var: ContextVar[str] = ContextVar('request_id',
  default='unknown')
session_id_var: ContextVar[str] = ContextVar('session_id',
  default='unknown')
user_id_var: ContextVar[str] = ContextVar('user_id',
  default='unknown')

def configure_logging():
structlog.configure(
  processors=[
 structlog.contextvars.merge_contextvars,
 structlog.processors.add_log_level,
 structlog.processors.TimeStamper(fmt="iso"),
 structlog.processors.JSONRenderer()
  ],
  wrapper_class=structlog.make_filtering_bound_logger(
  logging.INFO),
  context_class=dict,
  logger_factory=structlog.PrintLoggerFactory(),
  cache_logger_on_first_use=True,
)

logger = structlog.get_logger()

class AgentLogger:
@staticmethod
def log_llm_call(
  model: str,
  prompt_tokens: int,
  completion_tokens: int,
  latency_ms: float,
  cache_hit: bool = False
):
  logger.info(
 "llm_call",
 model=model,
 prompt_tokens=prompt_tokens,
```

```
     completion_tokens=completion_tokens,
     total_tokens=prompt_tokens + completion_tokens,
     latency_ms=latency_ms,
     cache_hit=cache_hit,
     request_id=request_id_var.get(),
     session_id=session_id_var.get(),
     user_id=user_id_var.get()
      )

    @staticmethod
    def log_tool_call(
      tool_name: str,
      tool_input: dict,
      tool_output: Any,
      success: bool,
      latency_ms: float,
      error: str = None
    ):
      logger.info(
     "tool_call",
     tool_name=tool_name,
     tool_input=tool_input,
     success=success,
     latency_ms=latency_ms,
     error=error,
     request_id=request_id_var.get(),
     session_id=session_id_var.get()
      )

    @staticmethod
    def log_agent_step(
      step_number: int,
      action_type: str,
      reasoning: str,
      outcome: str
    ):
      logger.info(
     "agent_step",
     step_number=step_number,
     action_type=action_type,
     reasoning=reasoning[:500],  # Truncate long reasoning
     outcome=outcome,
     request_id=request_id_var.get(),
     session_id=session_id_var.get()
      )

    @staticmethod
    def log_session_complete(
      total_steps: int,
      total_tokens: int,
      total_cost: float,
      success: bool,
      duration_seconds: float
    ):
      logger.info(
     "session_complete",
     total_steps=total_steps,
     total_tokens=total_tokens,
     total_cost=total_cost,
     success=success,
     duration_seconds=duration_seconds,
     request_id=request_id_var.get(),
     session_id=session_id_var.get(),
     user_id=user_id_var.get()
      )
```

Các Metric Quan trọng

Đây là các metric tôi theo dõi cho mọi hệ thống agent, với các triển khai Prometheus ví dụ:

```python
# agent_metrics.py
from prometheus_client import Counter, Histogram, Gauge

# Operational metrics
AGENT_REQUESTS = Counter(
'agent_requests_total',
'Total agent requests',
['agent_name', 'status']
)

AGENT_LATENCY = Histogram(
'agent_request_duration_seconds',
'Agent request latency',
['agent_name'],
buckets=[1, 5, 10, 30, 60, 120, 300, 600]
)

AGENT_STEPS = Histogram(
'agent_steps_per_request',
'Number of steps per agent request',
['agent_name'],
buckets=[1, 2, 5, 10, 20, 50, 100]
)

# Cost metrics
TOKEN_USAGE = Counter(
'agent_tokens_total',
'Total tokens used',
['agent_name', 'model', 'type']  # type: prompt/completion
)

API_COST = Counter(
'agent_api_cost_dollars',
'Total API cost in dollars',
['agent_name', 'model']
)

CACHE_HITS = Counter(
'agent_cache_hits_total',
'LLM cache hits',
['agent_name', 'cache_type']  # exact/semantic
)

# Quality metrics
TASK_SUCCESS = Counter(
'agent_task_success_total',
'Successful task completions',
['agent_name', 'task_type']
)

TASK_FAILURE = Counter(
'agent_task_failure_total',
'Failed task attempts',
['agent_name', 'task_type', 'failure_reason']
)

USER_FEEDBACK = Counter(
'agent_user_feedback_total',
'User feedback on agent responses',
['agent_name', 'rating']  # positive/negative
)

# Real-time gauges
ACTIVE_SESSIONS = Gauge(
```

```
        'agent_active_sessions',
        'Currently active agent sessions',
        ['agent_name']
        )

        QUEUE_DEPTH = Gauge(
        'agent_queue_depth',
        'Tasks waiting in queue',
        ['agent_name', 'priority']
        )
```

Các truy vấn Grafana dashboard ví dụ:

```
# Success rate over time
sum(rate(agent_task_success_total[5m])) /
sum(rate(agent_task_success_total[5m]) +
  rate(agent_task_failure_total[5m]))

# Cost per successful task
sum(rate(agent_api_cost_dollars[1h])) /
  sum(rate(agent_task_success_total[1h]))

# P95 latency
histogram_quantile(0.95,
  sum(rate(agent_request_duration_seconds_bucket[5m])) by (le))

# Cache efficiency
sum(rate(agent_cache_hits_total[5m])) /
sum(rate(agent_cache_hits_total[5m]) +
  rate(agent_tokens_total{type="prompt"}[5m]))
```

Cảnh báo Lỗi Agent

Thiết lập cảnh báo cho cả lỗi cấp tính và suy giảm dần:

```
        # agent-alerts.yaml
        groups:
          - name: agent-alerts
        rules:
         # Acute failures
         - alert: AgentHighErrorRate
          expr: |
        sum(rate(agent_task_failure_total[5m])) /
        sum(rate(agent_task_success_total[5m]) +
          rate(agent_task_failure_total[5m])) > 0.1
          for: 5m
          labels:
        severity: warning
          annotations:
        summary: "Agent error rate above 10%"

         - alert: AgentStuck
          expr: |
        agent_active_sessions > 0 and
        increase(agent_steps_per_request_count[10m]) == 0
          for: 10m
          labels:
        severity: critical
          annotations:
        summary: "Agent appears stuck - no progress in 10 minutes"

         # Cost alerts
         - alert: AgentCostSpike
          expr: |
        sum(rate(agent_api_cost_dollars[1h])) * 24 > 100
```

```
          for: 15m
          labels:
        severity: warning
          annotations:
        summary: "Agent projected daily cost exceeds $100"

         - alert: AgentTokenBurn
          expr: |
        sum(rate(agent_tokens_total[5m])) > 10000
          for: 5m
          labels:
        severity: warning
          annotations:
        summary: "Agent burning tokens at >10k/minute"

         # Quality degradation
         - alert: AgentQualityDrop
          expr: |
        sum(rate(agent_user_feedback_total{rating="positive"}[1h])) /
        sum(rate(agent_user_feedback_total[1h])) < 0.7
          for: 1h
          labels:
        severity: warning
          annotations:
        summary: "Agent positive feedback rate below 70%"

         # Latency degradation
         - alert: AgentSlowResponses
          expr: |
        histogram_quantile(0.95,
          sum(rate(agent_request_duration_seconds_bucket[5m])) by (le))
          > 120
          for: 10m
          labels:
        severity: warning
          annotations:
        summary: "Agent P95 latency exceeds 2 minutes"
```

Câu chuyện Thực chiến: Debug Agent lúc 3 Giờ Sáng

Chuông báo đến lúc 3:17 sáng. Agent infrastructure của chúng tôi, chịu trách nhiệm phản hồi các alert thông thường, đã trở nên điên loạn. Nó đang tạo hàng chục PagerDuty incident, mỗi cái đều tuyên bố đã "sửa" một vấn đề không tồn tại.

Tôi lảo đảo đến laptop và bắt đầu điều tra. Agent đang trong một vòng lặp: 1. Nhận một alert về CPU cao trên server X 2. "Sửa" nó bằng cách restart một service 3. Quan sát rằng CPU vẫn cao (vì đó là false alert) 4. Quyết định rằng cách sửa không hiệu quả 5. Escalate lên PagerDuty 6. Quay lại bước 2 với một "cách sửa" khác

Agent đã thử restart service, điều chỉnh resource limit, kill các process, và bây giờ đang cố gắng drain node—lúc 3 giờ sáng, không có sự giám sát của con người.

Vấn đề đầu tiên: Tôi không thể thấy agent đang *suy nghĩ* gì. Logging của chúng tôi ghi lại các hành động nhưng không ghi lại reasoning. Tôi phải tái tạo logic của nó từ chuỗi các lệnh nó chạy.

Vấn đề thứ hai: circuit breaker có bug. Nó kiểm tra tổng token nhưng không kiểm tra tổng *số hành động*. Agent đang thực hiện nhiều API call nhỏ, rẻ, ở dưới giới hạn token trong khi gây ra hỗn loạn.

Cách sửa tôi triển khai đêm đó:

```
# Enhanced circuit breaker
class AgentCircuitBreaker:
def __init__(self):
  self.max_tokens = 50000
  self.max_actions = 20
  self.max_duration = 300
  self.max_same_action = 3  # Don't repeat the same action
  self.action_history = []

def check(self, session: AgentSession, proposed_action: str) ->
  Tuple[bool, str]:
  # Token limit
  if session.total_tokens > self.max_tokens:
 return False, "Token limit exceeded"

  # Action limit
  if len(self.action_history) >= self.max_actions:
 return False, "Maximum actions reached"

  # Duration limit
  if session.duration_seconds > self.max_duration:
 return False, "Session timeout"

  # Repetition detection
  action_key = self._action_key(proposed_action)
  recent_same = sum(1 for a in self.action_history[-10:] if
  self._action_key(a) == action_key)
  if recent_same >= self.max_same_action:
 return False, f"Action '{action_key}' attempted too many times"

  self.action_history.append(proposed_action)
  return True, ""

def _action_key(self, action: str) -> str:
  """Extract key action type, ignoring parameters."""
  # "kubectl restart deployment/web" -> "kubectl restart"
  parts = action.split()
  return " ".join(parts[:2]) if len(parts) >= 2 else action
```

Và observability tốt hơn:

```
# Log reasoning, not just actions
def agent_step_with_reasoning(step_func):
@wraps(step_func)
def wrapper(self, *args, **kwargs):
  # Capture the agent's reasoning before acting
  reasoning = self.get_last_reasoning()

  AgentLogger.log_agent_step(
 step_number=self.current_step,
 action_type=self.pending_action.type,
 reasoning=reasoning,
 outcome="pending"
  )

  result = step_func(self, *args, **kwargs)

  AgentLogger.log_agent_step(
 step_number=self.current_step,
 action_type=self.pending_action.type,
 reasoning=reasoning,
 outcome="success" if result.success else f"failed:
  {result.error}"
  )
```

```
  return result
return wrapper
```

Bài học: khi debug agent lúc 3 giờ sáng, bạn cần hai thứ: khả năng thấy agent đang *suy nghĩ* gì, và khả năng ngăn nó làm những thứ ngu ngốc lặp đi lặp lại. Log reasoning. Đếm số hành động. Phát hiện các vòng lặp.

Chương 11: Các Mẫu Human-in-the-Loop

Tự động Hoàn toàn Hiếm khi là Mục tiêu

Khi tôi bắt đầu xây dựng AI agent, tôi có những viễn cảnh về các hệ thống hoàn toàn tự động sẽ xử lý mọi thứ mà không cần sự can thiệp của con người. Thực tế nhanh chóng sửa chữa ảo tưởng đó.

Tự động hoàn toàn có ba vấn đề lớn:

1. **Niềm tin cần thời gian để xây dựng.** Người dùng sẽ không tin tưởng một agent với các task quan trọng cho đến khi nó đã chứng minh bản thân ở những task nhỏ hơn.

2. **Các trường hợp biên là vô hạn.** Dù agent của bạn tốt đến đâu, nó sẽ gặp những tình huống mà nó không được chuẩn bị.

3. **Trách nhiệm quan trọng.** Khi có điều gì đó sai, “AI đã làm điều đó” không phải là câu trả lời chấp nhận được. Con người cần ở trong vòng lặp cho các quyết định quan trọng.

Mục tiêu không phải là loại bỏ con người khỏi vòng lặp—mà là đặt họ vào đúng những điểm trong vòng lặp, nơi sự phán đoán của họ thêm giá trị.

Quy trình Phê duyệt: Khi nào Dừng lại và Hỏi

Không phải tất cả các hành động đều bình đẳng. Một truy vấn read-only là rủi ro thấp; xóa database production thì không. Xây dựng agent của bạn với các cổng phê duyệt rõ ràng:

```
# approval_workflow.py
from enum import Enum
from dataclasses import dataclass
from typing import Optional, Callable
import asyncio

class RiskLevel(Enum):
LOW = "low"           # Read operations, safe queries
MEDIUM = "medium"     # Config changes, service restarts
HIGH = "high"         # Data modifications, deployments
CRITICAL = "critical" # Deletions, security changes

@dataclass
class ApprovalRequest:
request_id: str
action: str
risk_level: RiskLevel
reasoning: str
context: dict
timeout_seconds: int = 300
```

```
@dataclass
class ApprovalResponse:
approved: bool
approver: Optional[str]
comment: Optional[str]

class ApprovalGate:
def __init__(
  self,
  auto_approve_levels: list[RiskLevel] = [RiskLevel.LOW],
  notification_channel: str = "slack",
):
  self.auto_approve = set(auto_approve_levels)
  self.channel = notification_channel
  self.pending_approvals: dict[str, ApprovalRequest] = {}

async def request_approval(
  self,
  action: str,
  risk_level: RiskLevel,
  reasoning: str,
  context: dict
) -> ApprovalResponse:
  """Request approval for an action, blocking until approved or
  denied."""

  # Auto-approve low-risk actions
  if risk_level in self.auto_approve:
 return ApprovalResponse(
 approved=True,
 approver="auto",
 comment=f"Auto-approved (risk level: {risk_level.value})"
 )

  request = ApprovalRequest(
 request_id=generate_id(),
 action=action,
 risk_level=risk_level,
 reasoning=reasoning,
 context=context
  )

  self.pending_approvals[request.request_id] = request

  # Send notification
  await self._notify(request)

  # Wait for response
  try:
 response = await asyncio.wait_for(
 self._wait_for_response(request.request_id),
 timeout=request.timeout_seconds
 )
 return response
  except asyncio.TimeoutError:
 # Timeout = deny for safety
 return ApprovalResponse(
 approved=False,
 approver=None,
 comment="Request timed out - treating as denied"
 )
  finally:
 del self.pending_approvals[request.request_id]

async def _notify(self, request: ApprovalRequest):
  """Send approval request notification."""
  message = f"""
🤖 **Agent Approval Required**
```

```
**Action:** {request.action}
**Risk Level:** {request.risk_level.value.upper()}
**Reasoning:** {request.reasoning}

React with ✅ to approve or ❌ to deny.
Request ID: `{request.request_id}`
Timeout: {request.timeout_seconds}s
  """

  if self.channel == "slack":
 await slack_client.post_message(
 channel="#agent-approvals",
 text=message,
 metadata={"request_id": request.request_id}
 )
  elif self.channel == "pagerduty":
 # For critical actions, page someone
 await pagerduty_client.create_incident(
 title=f"Agent approval: {request.action}",
 body=message,
 urgency="high" if request.risk_level == RiskLevel.CRITICAL else
  "low"
 )
```

Cách sử dụng trong agent của bạn:

```
approval_gate = ApprovalGate(
auto_approve_levels=[RiskLevel.LOW, RiskLevel.MEDIUM]
)

async def execute_action(action: AgentAction) -> ActionResult:
risk = assess_risk(action)

if risk.level in [RiskLevel.HIGH, RiskLevel.CRITICAL]:
  approval = await approval_gate.request_approval(
 action=action.description,
 risk_level=risk.level,
 reasoning=action.reasoning,
 context={
 "command": action.command,
 "affected_resources": action.resources,
 "rollback_plan": action.rollback
 }
  )

  if not approval.approved:
 return ActionResult(
 success=False,
 error=f"Action denied by {approval.approver}:
  {approval.comment}"
 )

  # Log that this was human-approved
  audit_log.info(
 "action_approved",
 action=action.description,
 approver=approval.approver
  )

return await sandbox.execute(action)
```

Hàng đợi Review cho Output của Agent

Với các task mà agent tạo ra artifact (code, documentation, configuration), sử dụng hàng đợi review thay vì thực thi trực tiếp:

```
# review_queue.py
from dataclasses import dataclass
from datetime import datetime
from typing import List, Optional
import json

@dataclass
class ReviewItem:
id: str
agent_id: str
task_description: str
artifact_type: str  # code, config, documentation
artifact_content: str
reasoning: str
created_at: datetime
status: str = "pending"  # pending, approved, rejected, modified
reviewer: Optional[str] = None
review_comment: Optional[str] = None
reviewed_at: Optional[datetime] = None

class ReviewQueue:
def __init__(self, storage: Storage):
  self.storage = storage

async def submit_for_review(
  self,
  agent_id: str,
  task: str,
  artifact_type: str,
  content: str,
  reasoning: str
) -> str:
  """Submit an agent output for human review."""

  item = ReviewItem(
 id=generate_id(),
 agent_id=agent_id,
 task_description=task,
 artifact_type=artifact_type,
 artifact_content=content,
 reasoning=reasoning,
 created_at=datetime.utcnow()
  )

  await self.storage.save(item)
  await self._notify_reviewers(item)

  return item.id

async def get_pending(
  self,
  artifact_type: Optional[str] = None,
  limit: int = 50
) -> List[ReviewItem]:
  """Get pending items for review."""

  query = {"status": "pending"}
  if artifact_type:
 query["artifact_type"] = artifact_type

  return await self.storage.query(query, limit=limit)

async def approve(
  self,
  item_id: str,
  reviewer: str,
  comment: Optional[str] = None
) -> ReviewItem:
  """Approve an item and trigger execution."""
```

```
  item = await self.storage.get(item_id)
  item.status = "approved"
  item.reviewer = reviewer
  item.review_comment = comment
  item.reviewed_at = datetime.utcnow()

  await self.storage.save(item)

  # Trigger execution of approved item
  await self._execute_approved(item)

  return item

async def reject(
  self,
  item_id: str,
  reviewer: str,
  reason: str
) -> ReviewItem:
  """Reject an item with feedback."""

  item = await self.storage.get(item_id)
  item.status = "rejected"
  item.reviewer = reviewer
  item.review_comment = reason
  item.reviewed_at = datetime.utcnow()

  await self.storage.save(item)

  # Optionally, feed rejection back to agent for learning
  await self._record_feedback(item, positive=False)

  return item
```

Một CLI đơn giản để review:

```
# review_cli.py
import click
from rich.console import Console
from rich.syntax import Syntax
from rich.panel import Panel

console = Console()
queue = ReviewQueue(storage)

@click.command()
@click.option('--type', 'artifact_type', help='Filter by
  artifact type')
def review(artifact_type):
"""Interactive review of agent outputs."""

items = asyncio.run(queue.get_pending(
  artifact_type=artifact_type))

for item in items:
  console.clear()

  # Show task and reasoning
  console.print(Panel(
 f"[bold]Task:[/bold] {item.task_description}\n\n"
 f"[bold]Agent reasoning:[/bold]\n{item.reasoning}",
 title=f"Review Item: {item.id}",
 subtitle=f"Type: {item.artifact_type}"
  ))

  # Show artifact with syntax highlighting
  syntax = Syntax(
```

```
 item.artifact_content,
 lexer=get_lexer(item.artifact_type),
 theme="monokai",
 line_numbers=True
  )
  console.print(syntax)

  # Get reviewer decision
  console.print("\n[bold]Decision:[/bold]")
  console.print("  [green]a[/green] - Approve")
  console.print("  [red]r[/red] - Reject")
  console.print("  [yellow]s[/yellow] - Skip")
  console.print("  [blue]e[/blue] - Edit before approving")

  choice = click.prompt("Your choice", type=click.Choice(['a',
  'r', 's', 'e']))

  if choice == 'a':
 comment = click.prompt("Comment (optional)", default="")
 asyncio.run(queue.approve(item.id, get_current_user(), comment))
 console.print("[green]Approved![/green]")

  elif choice == 'r':
 reason = click.prompt("Reason for rejection")
 asyncio.run(queue.reject(item.id, get_current_user(), reason))
 console.print("[red]Rejected.[/red]")

  elif choice == 'e':
 edited = click.edit(item.artifact_content)
 if edited:
 item.artifact_content = edited
 asyncio.run(queue.approve(item.id, get_current_user(),
  "Approved with edits"))
```

Đường dẫn Escalation: Từ Agent đến Con người

Thiết kế các đường dẫn escalation rõ ràng khi agent bị kẹt hoặc không chắc chắn:

```
# escalation.py
from dataclasses import dataclass
from typing import Optional, List
from enum import Enum

class EscalationReason(Enum):
UNCERTAINTY = "uncertainty"      # Agent isn't confident
COMPLEXITY = "complexity"        # Task too complex
PERMISSION = "permission"        # Needs elevated access
FAILURE = "failure"              # Repeated failures
POLICY = "policy"                # Policy violation detected
USER_REQUEST = "user_request"    # User asked for human

@dataclass
class EscalationConfig:
uncertainty_threshold: float = 0.7  # Escalate if confidence
  below this
max_retries: int = 3                # Escalate after this many
  failures
max_complexity_score: int = 8       # Escalate for very complex
  tasks
escalation_channels: dict = None    # Maps reason -> channel

class EscalationManager:
def __init__(self, config: EscalationConfig):
  self.config = config
  self.channels = config.escalation_channels or {
 EscalationReason.UNCERTAINTY: "slack:#agent-help",
 EscalationReason.COMPLEXITY: "slack:#agent-help",
```

```
     EscalationReason.PERMISSION: "pagerduty:admin",
     EscalationReason.FAILURE: "slack:#agent-failures",
     EscalationReason.POLICY: "pagerduty:security",
     EscalationReason.USER_REQUEST: "slack:#agent-help"
      }

    def should_escalate(
      self,
      confidence: float,
      retry_count: int,
      complexity_score: int,
      policy_flags: List[str]
    ) -> Optional[EscalationReason]:
      """Check if current state warrants escalation."""

      if policy_flags:
     return EscalationReason.POLICY

      if confidence < self.config.uncertainty_threshold:
     return EscalationReason.UNCERTAINTY

      if retry_count >= self.config.max_retries:
     return EscalationReason.FAILURE

      if complexity_score > self.config.max_complexity_score:
     return EscalationReason.COMPLEXITY

      return None

    async def escalate(
      self,
      reason: EscalationReason,
      context: dict,
      session: AgentSession
    ) -> EscalationResult:
      """Escalate to humans and pause agent."""

      channel = self.channels.get(reason)

      message = self._format_escalation(reason, context, session)

      # Send to appropriate channel
      ticket_id = await self._send_escalation(channel, message)

      # Pause the agent session
      session.pause(
     reason=f"Escalated: {reason.value}",
     ticket_id=ticket_id
      )

      # Log escalation
      audit_log.info(
     "escalation",
     reason=reason.value,
     session_id=session.id,
     ticket_id=ticket_id
      )

      return EscalationResult(
     ticket_id=ticket_id,
     channel=channel,
     session_paused=True
      )

    def _format_escalation(
      self,
      reason: EscalationReason,
      context: dict,
      session: AgentSession
    ) -> str:
```

```
          return f"""
        🚨 **Agent Escalation**

        **Reason:** {reason.value}
        **Session:** {session.id}
        **User:** {session.user_id}

        **Current Task:**
        {session.current_task}

        **Agent's Last Reasoning:**
        {session.last_reasoning}

        **Context:**
```

{json.dumps(context, indent=2)}

```

**Session History:**
{self._format_history(session.action_history[-5:])}

**To Resume:**
Run `/agent resume {session.id}` after resolving.
  """
```

Tìm kiếm Sự Cân bằng Phù hợp về Quyền Tự động

Mức độ tự động phù hợp phụ thuộc vào nhiều yếu tố:

```
        # autonomy_levels.py
        from dataclasses import dataclass
        from typing import Dict, List

        @dataclass
        class AutonomyProfile:
        """Defines autonomy level for different contexts."""

        # What the agent can do without asking
        auto_approve: List[str]

        # What requires notification (but proceeds)
        notify_only: List[str]

        # What requires explicit approval
        require_approval: List[str]

        # What the agent cannot do at all
        forbidden: List[str]

        # Different profiles for different contexts
        AUTONOMY_PROFILES: Dict[str, AutonomyProfile] = {
        "development": AutonomyProfile(
          auto_approve=[
         "read_file", "list_directory", "run_tests",
         "format_code", "lint", "build",
         "git_status", "git_diff", "git_branch"
          ],
          notify_only=[
         "create_file", "modify_file", "delete_file",
         "git_commit", "install_dependency"
          ],
          require_approval=[
         "git_push", "deploy_staging", "modify_config"
          ],
          forbidden=[
```

```
 "deploy_production", "delete_database",
 "modify_infrastructure", "access_secrets"
  ]
),

"staging": AutonomyProfile(
  auto_approve=[
 "read_file", "list_directory", "view_logs",
 "check_metrics", "describe_resources"
  ],
  notify_only=[
 "restart_service", "scale_deployment",
 "run_migration_dry_run"
  ],
  require_approval=[
 "deploy", "run_migration", "modify_config",
 "change_resource_limits"
  ],
  forbidden=[
 "delete_resources", "access_production",
 "modify_security_groups"
  ]
),

"production": AutonomyProfile(
  auto_approve=[
 "read_file", "view_logs", "check_metrics",
 "describe_resources"
  ],
  notify_only=[
 # Almost nothing in prod should be silent
  ],
  require_approval=[
 "restart_service", "scale_deployment",
 "view_secrets", "run_query"
  ],
  forbidden=[
 "delete_anything", "modify_data",
 "deploy_without_approval", "access_pii"
  ]
),

"incident": AutonomyProfile(
  # During incidents, allow more autonomy for speed
  auto_approve=[
 "read_file", "view_logs", "check_metrics",
 "describe_resources", "restart_service",
 "scale_deployment", "rollback_deployment"
  ],
  notify_only=[
 "failover", "drain_node", "block_traffic"
  ],
  require_approval=[
 "modify_data", "delete_resources"
  ],
  forbidden=[
 "deploy_new_version"  # No deployments during incidents
  ]
)
}

class AutonomyManager:
def __init__(self, default_profile: str = "development"):
  self.current_profile = AUTONOMY_PROFILES[default_profile]

def set_profile(self, profile_name: str):
  if profile_name not in AUTONOMY_PROFILES:
 raise ValueError(f"Unknown profile: {profile_name}")
  self.current_profile = AUTONOMY_PROFILES[profile_name]
```

```
def check_action(self, action: str) -> ActionPermission:
  if action in self.current_profile.forbidden:
 return ActionPermission.FORBIDDEN
  if action in self.current_profile.require_approval:
 return ActionPermission.REQUIRE_APPROVAL
  if action in self.current_profile.notify_only:
 return ActionPermission.NOTIFY
  if action in self.current_profile.auto_approve:
 return ActionPermission.ALLOWED
  # Default: require approval for unknown actions
  return ActionPermission.REQUIRE_APPROVAL
```

Insight quan trọng: quyền tự động nên được kiếm được, không phải được cấp. Bắt đầu với các hạn chế chặt chẽ và nới lỏng chúng khi agent chứng minh bản thân. Và luôn luôn, luôn luôn có cách để rút phích cắm.

```
# The final safety net
class EmergencyStop:
"""Global emergency stop for all agents."""

def __init__(self, redis_client):
  self.redis = redis_client
  self.stop_key = "agent:emergency_stop"

def activate(self, reason: str, activated_by: str):
  """Stop all agents immediately."""
  self.redis.set(self.stop_key, json.dumps({
 "active": True,
 "reason": reason,
 "activated_by": activated_by,
 "activated_at": datetime.utcnow().isoformat()
  }))

  # Notify everyone
  alert_all_channels(
 f"🛑 EMERGENCY STOP ACTIVATED\n"
 f"Reason: {reason}\n"
 f"By: {activated_by}\n"
 f"All agents are now paused."
  )

def is_active(self) -> bool:
  data = self.redis.get(self.stop_key)
  if data:
 return json.loads(data).get("active", False)
  return False

def deactivate(self, deactivated_by: str):
  """Resume agent operations."""
  self.redis.delete(self.stop_key)
  alert_all_channels(
 f"✅ Emergency stop deactivated by {deactivated_by}\n"
 f"Agents may now resume."
  )

# Every agent checks this before every action
emergency = EmergencyStop(redis_client)

def execute_action(action: AgentAction):
if emergency.is_active():
  raise AgentPausedError("Emergency stop is active")
# ... proceed with action
```

Human-in-the-loop không phải là một hạn chế của hệ thống agent của bạn—đó là một tính năng. Đó là điều cho phép bạn deploy agent với sự tự tin, biết rằng khi (không phải nếu) có điều gì đó sai, có một con người sẵn sàng can thiệp.

Kết thúc Phần 3: Các Mẫu Thiết kế cho Production

Phần 4: Ứng Dụng Thực Tế

Chương 12: Tự Động Hóa Hạ Tầng

Lời hứa của tự động hóa hạ tầng luôn rất hấp dẫn: bạn khai báo những gì mình muốn, và máy móc sẽ tự tìm cách thực hiện. Chúng ta đã đi một chặng đường dài từ việc triển khai server thủ công đến Infrastructure as Code. Nhưng ngay cả với Terraform, Ansible và Kubernetes, vẫn còn một khoảng cách giữa "việc cần làm" và "người thực sự làm việc đó." AI agent đang lấp đầy khoảng trống này.

Agent Triển Khai và Quản Lý Hạ Tầng

Hãy bắt đầu với một ví dụ cụ thể. Đây là một agent triển khai hạ tầng dựa trên yêu cầu bằng ngôn ngữ tự nhiên:

```
from openai import OpenAI
import subprocess
import json

class InfrastructureAgent:
def __init__(self):
  self.client = OpenAI()
  self.tools = [
 {
 "type": "function",
 "function": {
     "name": "terraform_plan",
     "description": "Generate and show a Terraform plan",
     "parameters": {
         "type": "object",
         "properties": {
             "working_dir": {"type": "string"},
             "var_file": {"type": "string"}
         }
     }
 }
 },
 {
 "type": "function",
 "function": {
     "name": "terraform_apply",
     "description": "Apply Terraform changes (requires
  approval)",
     "parameters": {
         "type": "object",
         "properties": {
             "working_dir": {"type": "string"},
             "auto_approve": {"type": "boolean", "default":
  False}
         }
     }
 }
 },
```

```
         {
         "type": "function",
         "function": {
             "name": "generate_terraform",
             "description": "Generate Terraform configuration",
             "parameters": {
                 "type": "object",
                 "properties": {
                     "resource_type": {"type": "string"},
                     "config": {"type": "object"}
                 }
             }
         }
         }
          ]

        def execute_tool(self, tool_name, args):
          if tool_name == "terraform_plan":
         return subprocess.run(
         ["terraform", "plan", "-no-color"],
         cwd=args.get("working_dir", "."),
         capture_output=True,
         text=True
         ).stdout
          elif tool_name == "terraform_apply":
         if not args.get("auto_approve"):
         return "APPROVAL_REQUIRED: Run with auto_approve=true after
          human review"
         return subprocess.run(
         ["terraform", "apply", "-auto-approve", "-no-color"],
         cwd=args.get("working_dir", "."),
         capture_output=True,
         text=True
         ).stdout
          # ... additional tool implementations
```

Điểm mấu chốt ở đây không phải là code—mà là pattern. Agent chuyển đổi ý định cấp cao (“Tôi cần một môi trường staging giống production nhưng nhỏ hơn”) thành các hành động hạ tầng cụ thể. Nhưng hãy chú ý đến cổng `APPROVAL_REQUIRED`. Chúng ta không bao giờ trao quyền kiểm soát hạ tầng không giới hạn cho agent.

Hệ Thống Tự Phục Hồi: Phát Hiện và Sửa Lỗi

Đây là nơi agent thực sự tỏa sáng. Hãy xem cách tiếp cận truyền thống khi xử lý ổ đĩa đầy:

```
┌────────────────────────────────────────────────────────────┐
│                   Cách Tiếp Cận Truyền Thống               │
├────────────────────────────────────────────────────────────┤
│ 1. Alert kích hoạt: "Disk 95% full"                        │
│ 2. Kỹ sư trực ca bị đánh thức                              │
│ 3. SSH vào server                                          │
│ 4. Chạy df -h, du -sh để tìm file lớn                      │
│ 5. Kiểm tra cái nào có thể xóa an toàn                     │
│ 6. Dọn log, file tạm                                       │
│ 7. Ghi chép incident                                       │
│ 8. Quay lại ngủ (có thể)                                   │
└────────────────────────────────────────────────────────────┘
```

Còn đây là cách tiếp cận có agent hỗ trợ:

```
┌────────────────────────────────────────────────────────────┐
│                  Cách Tiếp Cận Có Agent Hỗ Trợ             │
```

```
├─────────────────────────────────────────────────────────────┤
│ 1. Alert kích hoạt: "Disk 95% full"                         │
│ 2. Agent nhận alert qua webhook                             │
│ 3. Agent SSH vào server, phân tích disk usage               │
│ 4. Agent xác định file có thể xóa an toàn dựa trên quy tắc: │
│    - Log cũ hơn 7 ngày                                      │
│    - File tạm trong /tmp                                    │
│    - Cache của package manager                              │
│ 5. Agent thực hiện dọn dẹp                                  │
│ 6. Agent xác minh disk đã dưới ngưỡng                       │
│ 7. Agent gửi tóm tắt lên Slack                              │
│ 8. Kỹ sư trực ca thức dậy thấy incident đã được giải quyết  │
└─────────────────────────────────────────────────────────────┘
```

Đây là một triển khai thực tế:

```python
class SelfHealingAgent:
"""Agent xử lý các vấn đề hạ tầng phổ biến một cách tự động."""

SAFE_CLEANUP_RULES = {
  "logs": {
 "paths": ["/var/log/*.log.*.gz", "/var/log/*.[0-9]"],
 "age_days": 7,
 "action": "delete"
  },
  "temp": {
 "paths": ["/tmp/*", "/var/tmp/*"],
 "age_days": 1,
 "action": "delete"
  },
  "apt_cache": {
 "paths": ["/var/cache/apt/archives/*.deb"],
 "action": "apt clean"
  },
  "journal": {
 "action": "journalctl --vacuum-time=7d"
  }
}

def handle_disk_alert(self, host: str, threshold: int = 90):
  """Xử lý alert về dung lượng disk."""

  # Bước 1: Phân tích tình huống
  disk_info = self.ssh_exec(host, "df -h / | tail -1")
  usage = self.parse_disk_usage(disk_info)

  if usage < threshold:
 return {"status": "already_resolved", "usage": usage}

  # Bước 2: Tìm các thư mục lớn nhất
  large_dirs = self.ssh_exec(host,
 "du -sh /* 2>/dev/null | sort -hr | head -10")

  # Bước 3: Thực hiện các quy tắc dọn dẹp an toàn
  freed_space = 0
  actions_taken = []

  for rule_name, rule in self.SAFE_CLEANUP_RULES.items():
 if "paths" in rule:
 for path_pattern in rule["paths"]:
     result = self.cleanup_path(host, path_pattern, rule)
     freed_space += result["freed_bytes"]
     actions_taken.append(result)
 elif "action" in rule:
 result = self.ssh_exec(host, rule["action"])
 actions_taken.append({"rule": rule_name, "result": result})

  # Bước 4: Xác minh cải thiện
```

```
          new_usage = self.parse_disk_usage(
         self.ssh_exec(host, "df -h / | tail -1")
          )

          # Bước 5: Báo cáo
          return {
         "status": "resolved" if new_usage < threshold else
          "needs_attention",
         "initial_usage": usage,
         "final_usage": new_usage,
         "freed_space": self.human_readable_size(freed_space),
         "actions": actions_taken
          }
```

Điểm hay của cách tiếp cận này là agent tuân theo các quy tắc được định nghĩa trước về những gì có thể chạm vào an toàn, nhưng sử dụng khả năng suy luận của AI để ưu tiên và xác minh các hành động. Nó không phải đang mù quáng chạy script—nó đang đưa ra quyết định trong các rào cản đã định.

Phát Hiện và Sửa Configuration Drift

Configuration drift là kẻ giết người thầm lặng của độ tin cậy hạ tầng. Bạn deploy một thứ gì đó, nó hoạt động, rồi ba tháng sau nó hỏng vì ai đó đã thay đổi thủ công một cài đặt mà Terraform không quản lý.

Đây là cách agent có thể giúp:

```
        class DriftDetectionAgent:
        """Phát hiện và tùy chọn sửa configuration drift."""

        def __init__(self, state_store: str):
          self.state_store = state_store
          self.known_state = self.load_known_state()

        def scan_for_drift(self, resources: list) -> dict:
          """So sánh state thực tế với state đã khai báo."""

          drift_report = {"drifted": [], "missing": [], "unexpected": []}

          for resource in resources:
         actual = self.get_actual_state(resource)
         expected = self.known_state.get(resource["id"])

         if expected is None:
         drift_report["unexpected"].append({
             "resource": resource,
             "actual": actual,
             "recommendation": "Import or delete"
         })
         elif actual is None:
         drift_report["missing"].append({
             "resource": resource,
             "expected": expected,
             "recommendation": "Recreate from state"
         })
         elif not self.states_match(actual, expected):
         drift_report["drifted"].append({
             "resource": resource,
             "actual": actual,
             "expected": expected,
             "diff": self.compute_diff(actual, expected),
             "recommendation": self.suggest_action(actual, expected)
```

```
     })

      return drift_report

    def auto_correct_drift(self, drift_report: dict,
                safe_only: bool = True) -> list:
      """Cố gắng sửa drift được phát hiện."""

      corrections = []

      for item in drift_report["drifted"]:
     if safe_only and not self.is_safe_correction(item):
     corrections.append({
         "resource": item["resource"]["id"],
         "status": "skipped",
         "reason": "Requires manual review"
     })
     continue

     # Tạo và thực hiện sửa chữa
     correction = self.generate_correction(item)
     result = self.execute_correction(correction)
     corrections.append({
     "resource": item["resource"]["id"],
     "status": "corrected" if result.success else "failed",
     "details": result
     })

      return corrections
```

Quy trình có agent hỗ trợ trông như thế này:

```
┌────────────────────────────────────────────────────────────────
  ──┐
│                 Quy Trình Phát Hiện Drift
  │
├────────────────────────────────────────────────────────────────
  ──┤
│
  │
│  ┌─────────┐      ┌─────────────┐      ┌─────────────┐
  │
│  │ Lịch    │──────│ Drift Scan  │──────│  Phân Loại  │
  │
│  │ (hourly)│      │   Agent     │      │   Drift     │
  │
│  └─────────┘      └─────────────┘      └──────┬──────┘
  │
│                                                │
  │
│
  ┌───────────────────────────┼───────────────┐  │
│                    │                           │
  │  │
│                    ▼                           ▼
  ▼  │
│            ┌─────────────┐           ┌─────────────┐
  ┌───────┐  │
│            │    An Toàn  │           │  Nguy Hiểm  │   │  Mới
  │  │
│            │ (auto-fix)  │           │(alert team) │
  │(review│  │
│            └──────┬──────┘           └──────┬──────┘
  └───┬───┘  │
│                   │                         │              │
  │
│                   ▼                         ▼              ▼
  │
```

```
│            ┌─────────────┐            ┌─────────────┐
  ┌───────┐  │
│            │   Áp Dụng   │            │    Slack    │   │  PR
  │  │
│            │   Sửa Chữa  │            │   Alert     │   │Tạo
 Mới│  │
│            └─────────────┘            └─────────────┘
  └───────┘  │
│
  │
└────────────────────────────────────────────────────────────────
 ──┘
```

Lập Kế Hoạch Công Suất với Sự Hỗ Trợ của AI

Lập kế hoạch công suất truyền thống bao gồm thu thập metrics, xây dựng bảng tính và đưa ra các phỏng đoán có cơ sở. AI agent có thể làm tốt hơn:

```
class CapacityPlanningAgent:
"""Lập kế hoạch công suất và đề xuất có sự hỗ trợ của AI."""

def analyze_capacity(self, service: str, timeframe_days: int =
  90):
  """Phân tích việc sử dụng tài nguyên và dự đoán nhu cầu tương
  lai."""

  # Thu thập dữ liệu lịch sử
  metrics = self.prometheus_query(f'''
 avg_over_time(
 container_cpu_usage_seconds_total{{service="{service}"}}
 [{timeframe_days}d:1h]
 )
  ''')

  memory_metrics = self.prometheus_query(f'''
 avg_over_time(
 container_memory_usage_bytes{{service="{service}"}}
 [{timeframe_days}d:1h]
 )
  ''')

  # Sử dụng LLM để phân tích xu hướng và tạo đề xuất
  prompt = f"""
  Analyze the following capacity metrics for service '{service}':

  CPU Usage (90 days):
  - Average: {metrics['avg']}%
  - Peak: {metrics['max']}%
  - Growth rate: {metrics['growth_rate']}% per month

  Memory Usage:
  - Average: {memory_metrics['avg']} GB
  - Peak: {memory_metrics['max']} GB
  - Growth rate: {memory_metrics['growth_rate']}% per month

  Current allocation:
  - CPU: {self.get_allocated('cpu', service)} cores
  - Memory: {self.get_allocated('memory', service)} GB

  Provide:
  1. Days until resource exhaustion at current growth rate
  2. Recommended new allocation
  3. Estimated cost impact
  4. Any anomalies or concerns observed
  """
```

```
        analysis = self.llm_analyze(prompt)

        return {
       "service": service,
       "current_metrics": {"cpu": metrics, "memory": memory_metrics},
       "analysis": analysis,
       "generated_at": datetime.utcnow().isoformat()
        }
```

Case Study: Agent Xử Lý Bảo Trì Định Kỳ

Hãy để tôi chia sẻ một triển khai thực tế mà chúng tôi đã chạy trong sáu tháng. Chúng tôi gọi nó là “MaintenanceBot.”

Vấn đề: Team của chúng tôi quản lý 47 server trên ba cloud provider. Mỗi tuần, ai đó phải: - Áp dụng các bản vá bảo mật - Rotate log không tự động rotate - Dọn dẹp Docker image bị bỏ rơi - Xác minh việc backup hoàn thành - Kiểm tra hết hạn certificate

Việc này tốn 4-6 tiếng mỗi tuần, và đó là công việc nhàm chán không ai muốn làm.

Giải pháp: Chúng tôi xây dựng một maintenance agent với kiến trúc sau:

```yaml
# maintenance_agent_config.yaml
name: MaintenanceBot
schedule: "0 3 * * 0"  # Mỗi Chủ nhật lúc 3 AM

tasks:
  - name: security_patches
type: package_update
scope: security_only
approval: auto  # security patches are pre-approved
rollback_on_failure: true

  - name: log_rotation
type: cleanup
patterns:
 - path: "/var/log/**/*.log"
  retention_days: 30
  compress: true

  - name: docker_cleanup
type: docker_prune
remove_unused_images: true
remove_dangling_volumes: true
min_age_hours: 48

  - name: backup_verification
type: verify
check: "backup_completed_within_24h"
alert_on_failure: true

  - name: certificate_check
type: monitor
check: "certificate_expiry_days > 30"
alert_threshold: 30
auto_renew: true  # Using certbot
```

Triển khai:

```
        class MaintenanceBot:
        def __init__(self, config_path: str):
          self.config = yaml.safe_load(open(config_path))
          self.inventory = self.load_inventory()
          self.llm = OpenAI()

        def run_maintenance_cycle(self):
          """Thực hiện chu kỳ bảo trì đầy đủ trên tất cả server."""

          results = {"servers": {}, "summary": {}}

          for server in self.inventory:
         server_results = {}

         for task in self.config["tasks"]:
         try:
             result = self.execute_task(server, task)
             server_results[task["name"]] = result
         except Exception as e:
             server_results[task["name"]] = {
                 "status": "error",
                 "error": str(e)
             }

             if task.get("rollback_on_failure"):
                 self.rollback(server, task)

         results["servers"][server["hostname"]] = server_results

          # Tạo tóm tắt dễ đọc
          results["summary"] = self.generate_summary(results)

          # Gửi lên Slack
          self.notify_team(results)

          return results

        def generate_summary(self, results: dict) -> str:
          """Sử dụng LLM để tạo tóm tắt dễ đọc."""

          prompt = f"""
          Generate a concise maintenance summary from these results:
          {json.dumps(results["servers"], indent=2)}

          Include:
          - Total servers maintained
          - Tasks completed vs failed
          - Any items requiring human attention
          - Disk space freed
          - Security patches applied

          Format for Slack (use emoji and formatting).
          """

          return self.llm.chat.completions.create(
         model="gpt-4",
         messages=[{"role": "user", "content": prompt}]
          ).choices[0].message.content
```

Kết quả sau 6 tháng:

Metric	Trước	Sau
Giờ bảo trì hàng tuần	4-6 tiếng	20 phút (chỉ review)
Chu kỳ vá bị bỏ lỡ	2-3/tháng	0

Metric	Trước	Sau
Alert về dung lượng disk	4-5/tháng	0
Incident hết hạn certificate	1 (đau đớn)	0

Agent không thay thế phán đoán của con người—chúng tôi vẫn review tóm tắt của nó mỗi sáng thứ Hai. Nhưng nó xử lý công việc tẻ nhạt một cách đáng tin cậy và nhất quán, điều mà thực tế con người không làm được.

Bài học quan trọng: 1. Bắt đầu với các thao tác chỉ đọc cho đến khi bạn tin tưởng agent 2. Mọi hành động phá hủy cần có logging và khả năng rollback 3. Việc tạo tóm tắt thực sự quan trọng—không có nó, không ai đọc báo cáo 4. Chế độ dry-run là thiết yếu để test thay đổi config

---

Chương 13: CI/CD và Deployment

Các pipeline Continuous Integration và Continuous Deployment là các động mạch của việc phát hành phần mềm hiện đại. Chúng cũng ngày càng phức tạp, dễ vỡ và tốn thời gian bảo trì. Đây chính xác là loại công việc nhàm chán mà agent xuất sắc trong việc loại bỏ.

Agent trong Pipeline Deployment

Pipeline CI/CD truyền thống là một chuỗi các bước cứng nhắc. Pipeline được tăng cường bởi agent thì có tính thích ứng:

```
┌────────────────────────────────────────────────────────────────
  ─┐
│        Pipeline Truyền Thống vs Được Tăng Cường bởi Agent
  │
├────────────────────────────────────────────────────────────────
  ─┤
│
  │
│  TRUYỀN THỐNG:
  │
│  ┌──────┐ ┌──────┐ ┌──────┐ ┌──────┐ ┌──────┐
  │
│  │Build │→│ Test │→│ Lint │→│Deploy│→│Verify│
  │
│  └──────┘ └──────┘ └──────┘ └──────┘ └──────┘
  │
│     ↓        ↓        ↓        ↓        ↓
  │
│   fail     fail     fail     fail     fail
  │
│     └────────┴────────┴────────┴────────┘
  │
│                       ↓
  │
│              [Con người điều tra]
  │
│
  │
│  ĐƯỢC TĂNG CƯỜNG BỞI AGENT:
  │
│  ┌──────┐ ┌──────┐ ┌──────┐ ┌──────┐ ┌──────┐
  │
│  │Build │→│ Test │→│ Lint │→│Deploy│→│Verify│
  │
│  └──────┘ └──────┘ └──────┘ └──────┘ └──────┘
  │
│     ↓        ↓        ↓        ↓        ↓
  │
│   fail     fail     fail     fail     fail
  │
│     └────────┴────────┴────────┴────────┘
  │
│                       ↓
  │
│              [Agent phân tích lỗi]
  │
│                       ↓
  │
```

```
│        ┌──────────────┬──────────────┐
  │
│        ↓              ↓              ↓
  │
│   [Tự sửa]    [Gợi ý sửa]  [Escalate]
  │
│
  │
└─────────────────────────────────────────────────────────────────
  ─┘
```

Đây là triển khai thực tế của một CI/CD assistant agent:

```python
class CICDAgent:
"""Agent hỗ trợ các hoạt động CI/CD pipeline."""

def __init__(self, github_token: str, slack_webhook: str):
  self.github = Github(github_token)
  self.slack_webhook = slack_webhook
  self.llm = OpenAI()

def analyze_failure(self, repo: str, run_id: int) -> dict:
  """Phân tích một CI run thất bại và xác định bước tiếp theo."""

  # Lấy log lỗi
  workflow_run = self.github.get_repo(repo).get_workflow_run(
  run_id)
  logs = self.fetch_logs(workflow_run)

  # Trích xuất thông tin lỗi liên quan
  errors = self.extract_errors(logs)

  # Lấy diff đã trigger run này
  diff = self.get_pr_diff(workflow_run)

  # Phân tích với LLM
  analysis_prompt = f"""
  Analyze this CI failure:

  ## Error logs:
  {errors}

  ## Code changes:
  {diff[:5000]}  # Truncate for context window

  Determine:
  1. Root cause of the failure
  2. Is this fixable automatically? (test fix, linting fix, etc.)
  3. Suggested fix (code or command)
  4. Confidence level (high/medium/low)

  Respond in JSON format.
  """

  analysis = self.llm.chat.completions.create(
 model="gpt-4-turbo",
 messages=[{"role": "user", "content": analysis_prompt}],
 response_format={"type": "json_object"}
  )

  result = json.loads(analysis.choices[0].message.content)
  result["workflow_run"] = run_id
  result["repository"] = repo

  return result

def attempt_fix(self, analysis: dict) -> dict:
  """Cố gắng sửa vấn đề nếu độ tin cậy đủ cao."""
```

```
 if analysis["confidence"] != "high":
return {
"action": "suggested",
"suggestion": analysis["suggested_fix"],
"reason": "Confidence too low for auto-fix"
}

 fix_type = analysis.get("fix_type")

 if fix_type == "test_fix":
return self.fix_failing_test(analysis)
 elif fix_type == "lint_fix":
return self.fix_lint_errors(analysis)
 elif fix_type == "dependency_fix":
return self.fix_dependency_issue(analysis)
 else:
return {
"action": "escalate",
"analysis": analysis,
"reason": f"Unknown fix type: {fix_type}"
}
```

Rollback Tự Động Dựa Trên Metrics

Một trong những tự động hóa CI/CD có giá trị nhất là rollback thông minh. Thay vì đợi con người nhận ra sự suy giảm, agent có thể theo dõi các metrics quan trọng và hành động:

```
class RollbackAgent:
"""Giám sát deployment và trigger rollback khi cần."""

ROLLBACK_THRESHOLDS = {
  "error_rate": {"warn": 0.01, "critical": 0.05},
  "latency_p99_ms": {"warn": 500, "critical": 1000},
  "success_rate": {"warn": 0.99, "critical": 0.95}
}

def monitor_deployment(self, service: str,
           deployment_id: str,
           duration_minutes: int = 15):
  """Giám sát deployment trong thời gian bake, rollback nếu phát
  hiện vấn đề."""

  start_time = datetime.utcnow()
  baseline = self.get_baseline_metrics(service)

  while datetime.utcnow() - start_time <
  timedelta(minutes=duration_minutes):
 current = self.get_current_metrics(service)

 # So sánh với baseline và các ngưỡng tuyệt đối
 issues = self.detect_issues(baseline, current)

 if issues["critical"]:
 self.execute_rollback(service, deployment_id, issues)
 return {
     "status": "rolled_back",
     "reason": issues["critical"],
     "deployment_id": deployment_id
 }

 if issues["warning"]:
 self.notify_team(f"Warning: {service} showing degradation",
  issues)
```

```
     time.sleep(30)  # Kiểm tra mỗi 30 giây

      return {"status": "success", "deployment_id": deployment_id}

    def execute_rollback(self, service: str, deployment_id: str,
      issues: dict):
      """Thực hiện rollback và thông báo cho team."""

      # Lấy phiên bản ổn định trước đó
      previous_version = self.get_previous_stable_version(service)

      # Thực hiện rollback
      if self.deployment_type == "kubernetes":
     result = subprocess.run([
     "kubectl", "rollout", "undo",
     f"deployment/{service}",
     "--namespace", self.namespace
     ], capture_output=True, text=True)
      elif self.deployment_type == "ecs":
     result = self.ecs_rollback(service, previous_version)

      # Ghi lại rollback
      self.record_rollback(service, deployment_id, previous_version,
      issues)

      # Thông báo cho team
      self.notify_team(
     f"🚨 Auto-rollback: {service}",
     f"Deployment {deployment_id} was rolled back due to:
      {issues['critical']}\n"
     f"Reverted to version: {previous_version}"
      )

      return result
```

Thông báo cho team bao gồm context mà con người cần để điều tra:

```
    def format_rollback_notification(self, service, deployment_id,
      issues, metrics):
    """Format một thông báo rollback chi tiết."""

    return f"""
    🚨 **Tự Động Rollback Đã Thực Hiện**

    **Service:** {service}
    **Deployment:** {deployment_id}
    **Thời gian:** {datetime.utcnow().isoformat()}

    **Điều Kiện Kích Hoạt:**
    {self.format_issues(issues)}

    **So Sánh Metrics:**
    | Metric | Baseline | Hiện tại | Ngưỡng |
    |--------|----------|---------|-----------|
    | Error Rate | {metrics['baseline']['error_rate']:.2%} |
      {metrics['current']['error_rate']:.2%} |
      {self.ROLLBACK_THRESHOLDS['error_rate']['critical']:.2%} |
    | P99 Latency | {metrics['baseline']['latency_p99_ms']}ms |
      {metrics['current']['latency_p99_ms']}ms |
      {self.ROLLBACK_THRESHOLDS['latency_p99_ms']['critical']}ms |

    **Hành Động Cần Thiết:**
    1. Review commit của deployment thất bại
    2. Kiểm tra regression trong test suite
    3. Validate fix trước khi re-deploy

    **Link Liên Quan:**
    - [Deployment Logs]({self.get_deployment_logs_url(
```

```
  deployment_id)})
- [Metrics Dashboard]({self.get_dashboard_url(service)})
- [Commit]({self.get_commit_url(deployment_id)})
"""
```

Hỗ Trợ Review PR

Code review tốn thời gian nhưng thiết yếu. Agent có thể cung cấp lượt review đầu tiên, bắt các vấn đề rõ ràng và để con người tập trung vào kiến trúc và logic:

```
class PRReviewAgent:
"""Trợ lý review PR tự động."""

def review_pr(self, repo: str, pr_number: int) -> dict:
  """Thực hiện review ban đầu của một pull request."""

  pr = self.github.get_repo(repo).get_pull(pr_number)
  files = pr.get_files()

  review_results = {
 "security": [],
 "performance": [],
 "style": [],
 "bugs": [],
 "suggestions": [],
 "questions": []
  }

  for file in files:
 if file.filename.endswith(('.py', '.js', '.ts', '.go')):
 file_review = self.review_file(file)
 for category, items in file_review.items():
     review_results[category].extend(items)

  # Kiểm tra test còn thiếu
  test_coverage = self.check_test_coverage(pr)
  if test_coverage["missing"]:
 review_results["suggestions"].append({
 "type": "missing_tests",
 "files": test_coverage["missing"],
 "message": "These files have changes but no corresponding test
  changes"
 })

  # Kiểm tra cập nhật documentation
  if self.needs_documentation_update(pr):
 review_results["suggestions"].append({
 "type": "documentation",
 "message": "This PR modifies public APIs but doesn't update
  documentation"
 })

  # Tạo tóm tắt
  review_results["summary"] =
  self.generate_review_summary(review_results)

  return review_results

def review_file(self, file) -> dict:
  """Review một file đơn lẻ để tìm vấn đề."""

  prompt = f"""
  Review this code change for a pull request:

  Filename: {file.filename}
```

```
          Changes:
```

{file.patch} ```

Look for: 1. Security issues (SQL injection, XSS, secrets in code, etc.) 2. Performance concerns (N+1 queries, unnecessary loops, etc.) 3. Potential bugs (null handling, edge cases, race conditions) 4. Style issues (naming, structure, readability)

For each issue found, provide: - Line number - Category - Severity (critical/high/medium/low) - Description - Suggested fix

Respond in JSON format. “””

response = self.llm.chat.completions.create(model=“gpt-4-turbo”, messages=[{“role”: “user”, “content”: prompt}], response_format={“type”: “json_object”})

return json.loads(response.choices[0].message.content)

~~~

Tự Động Hóa Phản Hồi Incident

Khi deployment gặp vấn đề, mỗi phút đều quan trọng. Agent có thể xử lý phản hồi ban đầu trong khi con
người đang được gọi:

```python
class IncidentResponseAgent:
"""First-responder tự động cho các incident deployment."""

def handle_incident(self, alert: dict) -> dict:
  """Tự động hóa phản hồi incident ban đầu."""

  incident = {
 "id": str(uuid.uuid4()),
 "started_at": datetime.utcnow().isoformat(),
 "alert": alert,
 "timeline": []
  }

  # Bước 1: Xác nhận và phân loại
  incident["timeline"].append({
 "time": datetime.utcnow().isoformat(),
 "action": "incident_started",
 "details": "Automated incident response initiated"
  })

  # Bước 2: Thu thập context
  context = self.gather_context(alert)
  incident["context"] = context
  incident["timeline"].append({
 "time": datetime.utcnow().isoformat(),
 "action": "context_gathered",
 "details": f"Gathered metrics, logs, and recent changes"
  })

  # Bước 3: Xác định các thay đổi gần đây
  recent_deploys = self.get_recent_deployments(
 context["service"],
 hours=2
  )
  incident["recent_deployments"] = recent_deploys

  # Bước 4: Tự động giảm thiểu nếu có thể
~~~

```
 if self.can_auto_mitigate(alert, context):
mitigation = self.execute_mitigation(alert, context)
incident["mitigation"] = mitigation
incident["timeline"].append({
"time": datetime.utcnow().isoformat(),
"action": "auto_mitigation",
"details": mitigation
})

 # Bước 5: Tạo incident channel và page on-call
 incident["channel"] = self.create_incident_channel(incident)
 self.page_oncall(incident)

 # Bước 6: Gửi tóm tắt ban đầu
 self.post_incident_summary(incident)

 return incident
```

Case Study: Agent Sửa Test Thất Bại

Đây là một trong những triển khai agent thành công nhất của chúng tôi. Hãy để tôi hướng dẫn bạn cách nó hoạt động.

Vấn đề: Test suite của chúng tôi có tỷ lệ flaky test 15%. Các kỹ sư thường chỉ chạy lại test thất bại, hy vọng chúng sẽ pass. Khi test thực sự thất bại, mất trung bình 45 phút để chẩn đoán và sửa các vấn đề đơn giản.

Giải pháp: Chúng tôi xây dựng một test-fixing agent mà: 1. Phát hiện test thất bại 2. Phân tích pattern thất bại 3. Cố gắng sửa các vấn đề phổ biến 4. Tạo PR cho các sửa chữa thành công

```python
class TestFixerAgent:
"""Agent tự động sửa các lỗi test phổ biến."""

FIXABLE_PATTERNS = [
  "assertion_error",
  "timeout",
  "import_error",
  "fixture_missing",
  "mock_misconfigured",
  "async_timing"
]

def analyze_test_failure(self, test_file: str,
             test_name: str,
             error_output: str) -> dict:
  """Phân tích một test thất bại và xác định có thể sửa được
  không."""

  # Đọc file test
  test_code = open(test_file).read()

  # Đọc code đang được test
  tested_module = self.get_tested_module(test_file)
  tested_code = open(tested_module).read() if tested_module else
  ""

  prompt = f"""
  Analyze this test failure:

  ## Test file: {test_file}
```

~~~
          ```python
          {test_code}
~~~

Tested module: {tested_module} `python   {tested_code[:3000]}`

Error output: `{error_output}`

Determine: 1. Failure category: {self.FIXABLE_PATTERNS} 2. Root cause 3. Is the test wrong, or is the code wrong? 4. If test is wrong, provide the fix 5. Confidence (high/medium/low)

Important: Only suggest test fixes, never production code fixes. “””

return self.llm_analyze(prompt)

def fix_test(self, analysis: dict, test_file: str) -> dict: “““Áp dụng sửa chữa được gợi ý và xác minh nó hoạt động.”””

if analysis[“confidence”] != “high”: return {“status”: “skipped”, “reason”: “Low confidence”}

if analysis[“fix_target”] != “test”: return {“status”: “skipped”, “reason”: “Would modify production code”}

Tạo branch branch_name = f”auto-fix/{analysis[‘test_name’]}-{uuid.uuid4( ).hex[:8]}” self.git_create_branch(branch_name)

Áp dụng sửa chữa original_content = open(test_file).read() fixed_content = self.apply_fix(original_content, analysis[“fix”])

with open(test_file, ‘w’) as f: f.write(fixed_content)

Chạy test cụ thể result = subprocess.run([“pytest”, test_file, “-k”, analysis[“test_name”], “-v”], capture_output=True, text=True)

if result.returncode == 0: # Test pass - tạo PR self.git_commit(test_file, f”fix: auto-fix {analysis[‘test_name’]}“) self.git_push(branch_name) pr = self.create_pull_request(branch_name, analysis)

return { “status”: “fixed”, “branch”: branch_name, “pr_url”: pr.html_url, “analysis”: analysis } else: # Sửa không hoạt động - rollback with open(test_file, ‘w’) as f: f.write(original_content) self.git_delete_branch(branch_name)

return { “status”: “failed”, “reason”: “Fix didn’t make test pass”, “output”: result.stdout + result.stderr }

~~~

Ví dụ sửa chữa thực tế:

Đây là một test thất bại thực tế mà agent đã sửa:

```python
# Test thất bại ban đầu
def test_user_creation_timestamp():
user = User.create(name="Test")
assert user.created_at == datetime.now()  # Fail do timing

# Sửa chữa của Agent
~~~

```
def test_user_creation_timestamp():
before = datetime.now()
user = User.create(name="Test")
after = datetime.now()
assert before <= user.created_at <= after  # Xử lý timing đúng
  cách
```

Kết quả sau 3 tháng:

Metric	Trước	Sau
Thời gian trung bình sửa lỗi test đơn giản	45 phút	2 phút (tự động)
Test được tự động sửa mỗi tuần	0	12-15
Giờ developer tiết kiệm mỗi tuần	0	~8 tiếng
Tỷ lệ false positive	N/A	3% (dễ reject)

Các ràng buộc quan trọng chúng tôi học được cần triển khai: 1. **Không bao giờ sửa production code** - Chỉ file test 2. **Yêu cầu con người merge** - PR cần approval, ngay cả những cái rõ ràng 3. **Một sửa chữa mỗi PR** - Dễ review và revert hơn 4. **Bao gồm phân tích** - PR description giải thích lý do 5. **Tag các pattern flaky** - Xây dựng database các vấn đề phổ biến

---

Chương 14: Observability và Phản Hồi Incident

Lúc 3 giờ sáng, khi pager của bạn kêu, điều cuối cùng bạn muốn làm là lọc qua hàng nghìn dòng log để cố hiểu chuyện gì đang xảy ra. Đây là nơi AI agent có thể là đồng minh lớn nhất của bạn—không thay thế chuyên môn của bạn, mà khuếch đại nó.

Phân Tích Log Ở Quy Mô Lớn

Các ứng dụng hiện đại tạo ra lượng log khổng lồ. Agent có thể là tuyến phân tích đầu tiên của bạn:

```
class LogAnalysisAgent:
"""Agent cho phân tích log thông minh và phát hiện pattern."""

def __init__(self, elasticsearch_client, llm_client):
  self.es = elasticsearch_client
  self.llm = llm_client
  self.known_patterns = self.load_known_patterns()

def analyze_timerange(self, service: str,
          start_time: datetime,
          end_time: datetime) -> dict:
  """Phân tích log cho một khoảng thời gian, làm nổi bật các
  pattern quan trọng."""

  # Query log
  logs = self.es.search(
 index=f"logs-{service}-*",
 body={
 "query": {
     "bool": {
         "must": [
             {"range": {"@timestamp": {
                 "gte": start_time.isoformat(),
                 "lte": end_time.isoformat()
             }}}
         ]
     }
 },
 "size": 10000,
 "sort": [{"@timestamp": "asc"}]
 }
  )

  # Trích xuất pattern
  error_patterns = self.extract_error_patterns(logs)
  anomalies = self.detect_anomalies(logs)

  # Phân tích với LLM
  analysis = self.llm_analyze_logs(logs, error_patterns,
  anomalies)

  return {
 "service": service,
 "timerange": {"start": start_time, "end": end_time},
 "log_count": len(logs["hits"]["hits"]),
```

```
 "error_patterns": error_patterns,
 "anomalies": anomalies,
 "analysis": analysis
  }

def llm_analyze_logs(self, logs: dict,
         errors: list,
         anomalies: list) -> dict:
  """Sử dụng LLM để tạo insight từ các pattern log."""

  # Lấy mẫu log thông minh - bao gồm lỗi và context
  sampled_logs = self.intelligent_sample(logs, errors)

  prompt = f"""
  Analyze these application logs and identify:
  1. Root cause of any errors
  2. Sequence of events leading to issues
  3. Correlations between different log entries
  4. Recommended investigation steps

  ## Error patterns found:
  {json.dumps(errors, indent=2)}

  ## Anomalies detected:
  {json.dumps(anomalies, indent=2)}

  ## Sample logs with context:
  {sampled_logs}

  Provide analysis in JSON format with:
  - summary: one-sentence description
  - root_cause: likely root cause
  - timeline: sequence of events
  - affected_components: list of impacted parts
  - recommended_actions: prioritized list of next steps
  """

  response = self.llm.chat.completions.create(
 model="gpt-4-turbo",
 messages=[{"role": "user", "content": prompt}],
 response_format={"type": "json_object"}
  )

  return json.loads(response.choices[0].message.content)
```

Thực Thi Runbook Tự Động

Runbook là các bước được ghi chép để xử lý các vấn đề đã biết. Agent có thể thực thi chúng tự động:

```
# runbook_definitions.yaml
runbooks:
  - id: high_memory_usage
trigger:
 metric: container_memory_usage_percent
 condition: "> 90"
 duration: 5m
steps:
 - name: gather_info
  action: exec
  command: "kubectl top pods -n {{namespace}}"

 - name: check_for_leaks
  action: exec
  command: "kubectl exec {{pod}} -- cat /proc/meminfo"
```

```
 - name: check_heap_dump
  action: conditional
  condition: "{{memory_growth_rate}} > 10"
  then:
- action: exec
 command: "kubectl exec {{pod}} -- jcmd 1 GC.heap_dump
  /tmp/heap.hprof"
- action: copy
 from: "{{pod}}:/tmp/heap.hprof"
 to: "s3://incident-artifacts/{{incident_id}}/heap.hprof"

 - name: restart_if_critical
  action: conditional
  condition: "{{memory_percent}} > 95"
  approval: required
  then:
- action: exec
 command: "kubectl rollout restart deployment/{{service}} -n
  {{namespace}}"

  - id: database_connection_exhausted
trigger:
 log_pattern: "connection pool exhausted"
 count: 5
 window: 1m
steps:
 - name: check_connection_count
  action: exec
  command: "psql -c 'SELECT count(*) FROM pg_stat_activity'"

 - name: identify_long_queries
  action: exec
  command: |
psql -c "SELECT pid, now() - pg_stat_activity.query_start AS
  duration, query
    FROM pg_stat_activity
    WHERE state != 'idle'
    ORDER BY duration DESC
    LIMIT 10"

 - name: kill_long_queries
  action: conditional
  condition: "any(query.duration > '10 minutes')"
  approval: required
  then:
- action: exec
 command: "psql -c 'SELECT pg_terminate_backend({{pid}})'"
```

Đây là agent thực thi các runbook này:

```
class RunbookAgent:
"""Thực thi các runbook được định nghĩa trước để phản hồi
  incident."""

def __init__(self, runbook_path: str):
  self.runbooks = yaml.safe_load(open(runbook_path))
  self.execution_history = []

def execute_runbook(self, runbook_id: str, context: dict) ->
  dict:
  """Thực thi một runbook với context đã cho."""

  runbook = self.find_runbook(runbook_id)
  execution = {
 "runbook_id": runbook_id,
 "started_at": datetime.utcnow().isoformat(),
 "context": context,
```

```
 "steps": []
  }

  for step in runbook["steps"]:
 step_result = self.execute_step(step, context)
 execution["steps"].append(step_result)

 # Cập nhật context với kết quả của step
 context[step["name"]] = step_result

 # Dừng nếu step yêu cầu approval
 if step_result.get("awaiting_approval"):
 execution["status"] = "awaiting_approval"
 execution["pending_step"] = step["name"]
 return execution

 # Dừng nếu step thất bại và runbook không cho phép tiếp tục
 if step_result["status"] == "failed" and not
  step.get("continue_on_failure"):
 execution["status"] = "failed"
 execution["failed_step"] = step["name"]
 return execution

  execution["status"] = "completed"
  execution["completed_at"] = datetime.utcnow().isoformat()

  return execution

def execute_step(self, step: dict, context: dict) -> dict:
  """Thực thi một step runbook đơn lẻ."""

  action = step["action"]

  if action == "exec":
 command = self.interpolate(step["command"], context)
 result = subprocess.run(
 command, shell=True, capture_output=True, text=True
 )
 return {
 "name": step["name"],
 "status": "success" if result.returncode == 0 else "failed",
 "output": result.stdout,
 "error": result.stderr
 }

  elif action == "conditional":
 condition_met = self.evaluate_condition(step["condition"],
  context)
 if condition_met:
 if step.get("approval") == "required":
     return {
         "name": step["name"],
         "status": "pending",
         "awaiting_approval": True,
         "condition_met": True,
         "proposed_actions": step["then"]
     }
 # Thực thi các step 'then'
 for sub_step in step["then"]:
     self.execute_step(sub_step, context)
 return {
 "name": step["name"],
 "status": "success",
 "condition_met": condition_met
 }

  # ... xử lý các action type khác
```

Tóm Tắt Incident và Soạn Thảo RCA

Sau khi mọi thứ lắng xuống, ai đó phải viết báo cáo sau incident. Agent có thể soạn thảo nó:

```
class IncidentSummarizationAgent:
"""Soạn thảo báo cáo sau incident từ dữ liệu incident."""

def generate_postmortem(self, incident: dict) -> str:
  """Tạo bản thảo tài liệu post-mortem."""

  # Thu thập tất cả dữ liệu liên quan
  timeline = self.build_timeline(incident)
  metrics = self.gather_metrics(incident)
  logs = self.gather_relevant_logs(incident)
  actions_taken = incident.get("actions", [])

  prompt = f"""
  Generate a post-incident report (postmortem) from this
  incident data:

  ## Incident Overview
  - ID: {incident['id']}
  - Started: {incident['started_at']}
  - Resolved: {incident.get('resolved_at', 'Ongoing')}
  - Severity: {incident['severity']}
  - Affected Services: {', '.join(incident['affected_services'])}

  ## Timeline
  {json.dumps(timeline, indent=2)}

  ## Key Metrics During Incident
  {json.dumps(metrics, indent=2)}

  ## Actions Taken
  {json.dumps(actions_taken, indent=2)}

  ## Relevant Log Excerpts
  {logs}

  Generate a postmortem with these sections:
  1. Executive Summary (2-3 sentences)
  2. Impact (users affected, duration, business impact)
  3. Timeline (key events with timestamps)
  4. Root Cause Analysis (what went wrong and why)
  5. Resolution (how it was fixed)
  6. Action Items (preventive measures, categorized as P0/P1/P2)
  7. Lessons Learned

  Write in a blameless tone. Focus on systems and processes, not
  individuals.
  """

  response = self.llm.chat.completions.create(
 model="gpt-4-turbo",
 messages=[{"role": "user", "content": prompt}]
  )

  return response.choices[0].message.content
```

Hỗ Trợ On-Call

Trải nghiệm on-call là nơi agent có thể có tác động lớn nhất đến chất lượng cuộc sống:

```
class OnCallAssistantAgent:
"""Trợ lý AI cho kỹ sư on-call."""

def handle_alert(self, alert: dict) -> dict:
  """Xử lý alert đến và cung cấp context."""

  # Làm giàu alert với context
  context = {
 "alert": alert,
 "service_info": self.get_service_info(alert["service"]),
 "recent_deployments":
  self.get_recent_deployments(alert["service"]),
 "similar_incidents": self.find_similar_incidents(alert),
 "relevant_runbooks": self.find_relevant_runbooks(alert),
 "current_metrics": self.get_current_metrics(alert["service"]),
 "on_call_roster": self.get_on_call_roster(alert["service"])
  }

  # Tạo phân tích ban đầu
  analysis = self.analyze_alert(context)

  # Format cho Slack
  message = self.format_oncall_message(alert, context, analysis)

  return {
 "context": context,
 "analysis": analysis,
 "suggested_actions": analysis.get("suggested_actions", []),
 "confidence": analysis.get("confidence", "medium"),
 "slack_message": message
  }

def answer_question(self, question: str, incident_context: dict)
  -> str:
  """Trả lời câu hỏi của kỹ sư on-call về incident."""

  prompt = f"""
  You are an AI assistant helping an on-call engineer during an
  incident.

  ## Current Incident Context
  {json.dumps(incident_context, indent=2)}

  ## Engineer's Question
  {question}

  Provide a helpful, accurate answer. If you don't know
  something, say so.
  Include relevant commands or queries when appropriate.
  """

  response = self.llm.chat.completions.create(
 model="gpt-4-turbo",
 messages=[{"role": "user", "content": prompt}]
  )

  return response.choices[0].message.content
```

Case Study: Agent Là First Responder Của Bạn

Hãy để tôi chia sẻ agent có tác động lớn nhất chúng tôi đã xây dựng: một incident first-responder đã giảm mean time to resolution (MTTR) của chúng tôi 40%.

Vấn đề: On-call rotation của chúng tôi có 6 kỹ sư. Thời gian phản hồi rất khác nhau—từ 2 phút cho người có kinh nghiệm nhất đến 30+ phút cho thành viên mới hơn. 15 phút đầu của bất kỳ incident nào thường được dùng để thu thập context.

Giải pháp: Một agent mà: 1. Nhận mọi alert 2. Ngay lập tức thu thập context liên quan 3. Gửi một briefing toàn diện đến incident channel 4. Gợi ý các bước điều tra ban đầu 5. Có thể trả lời câu hỏi từ kỹ sư on-call

Đây là kiến trúc:

```
┌────────────────────────────────────────────────────────────────
  ─────┐
│                    Kiến Trúc First Responder Agent
  │
├────────────────────────────────────────────────────────────────
  ─────┤
│
  │
│  ┌──────────┐     ┌──────────────────┐
  ┌─────────────────────┐  │
│  │  Alert   │────►│  First Responder │────►│  Incident Channel
  │  │
│  │(PagerDuty)    │      Agent         │     │     (Slack)
  │  │
│  └──────────┘     └────────┬─────────┘
  └──────────┬──────────┘  │
│                            │                          │
  │
│                            ▼                          │
  │
│  ┌─────────────────────────────────────────────┐     │
  │
│  │           Thu Thập Context                   │     │
  │
│  │  ┌─────────┐ ┌─────────┐ ┌─────────────┐   │     │
  │
│  │  │Prometheus│ │Elastic- │ │   GitHub    │   │     │
  │
│  │  │ Metrics  │ │search   │ │  Deployments│   │     │
  │
│  │  └─────────┘ └─────────┘ └─────────────┘   │     │
  │
│  │  ┌─────────┐ ┌─────────┐ ┌─────────────┐   │     │
  │
│  │  │Incident │ │Runbook  │ │   Service   │   │     │
  │
│  │  │Lịch Sử  │ │  DB     │ │   Catalog   │   │     │
  │
│  │  └─────────┘ └─────────┘ └─────────────┘   │     │
  │
│  └─────────────────────────────────────────────┘     │
  │
│                                                       │
  │
│                          ▼                            ▼
  │
│                 ┌─────────────────────────────────────────┐
  │
│                 │       Kỹ Sư On-Call                      │
  │
│                 │  • Thấy đầy đủ context ngay lập tức      │
  │
│                 │  • Có thể hỏi agent câu hỏi              │
  │
│                 │  • Có runbook được gợi ý sẵn sàng        │
```

```
  │
│                 └──────────────────────────────────────────┘
  │
│
  │
└─────────────────────────────────────────────────────────────────
  ──────┘
```

Triển khai:

```
class FirstResponderAgent:
"""
First responder được hỗ trợ bởi AI cho incident.
Thu thập context và cung cấp briefing trong vài giây sau alert.
"""

def __init__(self):
  self.prometheus = PrometheusClient()
  self.elasticsearch = ElasticsearchClient()
  self.github = GithubClient()
  self.pagerduty = PagerDutyClient()
  self.slack = SlackClient()
  self.llm = OpenAI()
  self.vector_store = VectorStore()  # Cho tìm kiếm incident
  tương tự

def on_alert(self, alert: dict):
  """Xử lý alert đến - điểm vào chính."""

  start_time = time.time()

  # Tạo incident channel
  channel = self.slack.create_channel(
 f"inc-{alert['service']}-{datetime.now().strftime('%m%d-%H%M')}"
  )

  # Thu thập context song song
  with ThreadPoolExecutor(max_workers=6) as executor:
 futures = {
 'metrics': executor.submit(self.get_metrics_context, alert),
 'logs': executor.submit(self.get_log_context, alert),
 'deployments': executor.submit(self.get_deployment_context,
  alert),
 'similar': executor.submit(self.find_similar_incidents, alert),
 'runbooks': executor.submit(self.find_runbooks, alert),
 'service_info': executor.submit(self.get_service_info, alert)
 }

 context = {k: f.result() for k, f in futures.items()}

  # Tạo phân tích
  analysis = self.analyze_incident(alert, context)

  # Gửi briefing
  briefing = self.format_briefing(alert, context, analysis)
  self.slack.post_message(channel, briefing)

  # Log timing
  elapsed = time.time() - start_time
  self.log_metric('first_response_time_seconds', elapsed)

  # Trả về cho xử lý tiếp theo có thể
  return {
 "channel": channel,
 "context": context,
 "analysis": analysis,
 "response_time_seconds": elapsed
  }
```

```
    def format_briefing(self, alert, context, analysis) -> str:
      """Format incident briefing cho Slack."""

      return f"""
    🚨 *Incident Alert: {alert['name']}*
    *Service:* {alert['service']} | *Severity:* {alert['severity']}
      | *Time:* {alert['timestamp']}

    *📊 Trạng Thái Nhanh:*
    • Error rate: {context['metrics']['error_rate']:.2%} (bình
      thường: <1%)
    • P99 latency: {context['metrics']['latency_p99']}ms (bình
      thường: <200ms)
    • Active instance: {context['metrics']['instance_count']}

    *🔍 Phân Tích AI:*
    {analysis['summary']}

    *Nguyên nhân có thể:* {analysis['likely_cause']}
    *Độ tin cậy:* {analysis['confidence']}

    *📦 Thay Đổi Gần Đây:*
    {self.format_deployments(context['deployments'])}

    *📚 Incident Tương Tự Trước Đây:*
    {self.format_similar_incidents(context['similar'])}

    *📋 Runbook Được Gợi Ý:*
    {self.format_runbooks(context['runbooks'])}

    *💡 Các Bước Đầu Tiên Được Gợi Ý:*
    {self.format_suggested_actions(analysis['suggested_actions'])}

    ---
    _Hỏi tôi câu hỏi trong channel này - tôi ở đây để giúp điều tra._
    _Gõ `/incident-action` để thực hiện các bước runbook._
    """

    def answer_question(self, channel: str, question: str):
      """Trả lời câu hỏi từ kỹ sư on-call trong incident channel."""

      # Lấy incident context từ channel
      incident = self.get_incident_from_channel(channel)

      # Xây dựng context hội thoại
      messages = [
     {"role": "system", "content": """
     You are an AI assistant helping with an active incident.
     You have access to logs, metrics, and service documentation.
     Be concise, accurate, and helpful. If you're not sure, say so.
     Include specific commands, queries, or links when relevant.
     """},
     {"role": "user", "content": f"""
     Incident context: {json.dumps(incident, indent=2)}

     Engineer's question: {question}
     """}
      ]

      response = self.llm.chat.completions.create(
     model="gpt-4-turbo",
     messages=messages
      )

      self.slack.post_message(channel,
      response.choices[0].message.content)
```

Ví dụ incident thực tế:

```
🚨 Incident Alert: High Error Rate - Payment Service
Service: payment-service | Severity: SEV1 | Time: 2024-01-15
  03:42 UTC

📊 Trạng Thái Nhanh:
• Error rate: 23.4% (bình thường: <1%)
• P99 latency: 2,340ms (bình thường: <200ms)
• Active instance: 3/3

🔍 Phân Tích AI:
Error spike tương quan với deployment 15 phút trước. Log cho thấy
lỗi "connection refused" đến payment-gateway.internal. Phiên bản
  mới
đường như có service endpoint bị cấu hình sai.

Nguyên nhân có thể: Lỗi cấu hình trong deployment v2.34.1 - sai
  payment
gateway endpoint.
Độ tin cậy: Cao

📦 Thay Đổi Gần Đây:
• v2.34.1 deployed 15 phút trước bởi @alice - "Update payment
  gateway integration"
• Không có thay đổi infrastructure trong 24h qua

📚 Incident Tương Tự Trước Đây:
• INC-2023-0892 (3 tháng trước): Payment service config error -
  giải quyết bằng rollback
• INC-2023-0654 (6 tháng trước): Gateway connectivity - giải
  quyết bằng config fix

📋 Runbook Được Gợi Ý:
• [RB-PAY-001] Payment Service Rollback
• [RB-PAY-003] Payment Gateway Connectivity Check

💡 Các Bước Đầu Tiên Được Gợi Ý:
1. Kiểm tra payment gateway endpoint trong config hiện tại
   kubectl get configmap payment-config -o yaml
2. So sánh với phiên bản trước
   kubectl rollout history deployment/payment-service
3. Nếu config sai, rollback ngay lập tức
   kubectl rollout undo deployment/payment-service
```

Kết quả sau 6 tháng:

Metric	Trước	Sau	Cải Thiện
Mean Time to Resolution	47 phút	28 phút	Nhanh hơn 40%
Thời gian phản hồi đầu tiên	8 phút	45 giây	Nhanh hơn 90%
Thời gian thu thập context	15 phút	0 (tự động)	Loại bỏ
MTTR của kỹ sư junior	65 phút	32 phút	Nhanh hơn 51%
Tỷ lệ escalate incident	34%	22%	Giảm 35% escalation

Các quyết định thiết kế quan trọng đã làm cho điều này hoạt động:

1. **Tốc độ quan trọng nhất** - Mọi thứ chạy song song. Agent phải gửi trong vòng 60 giây.

2. **Context hơn command** - Agent cung cấp context và gợi ý, không phải command. Con người đưa ra quyết định.

3. **Học từ lịch sử** - Tìm kiếm vector similarity với các incident trước đây hiệu quả đáng ngạc nhiên.

4. **Giao diện hội thoại** - Kỹ sư có thể hỏi câu hỏi tiếp theo một cách tự nhiên.

5. **Tích hợp là tất cả** - Agent pull từ 8 hệ thống khác nhau. Mỗi tích hợp đều tốn công sức, nhưng đáng giá liên tục.

6. **Degradation uyển chuyển** - Nếu bất kỳ nguồn context nào fail, agent vẫn gửi những gì nó có.

Những gì chúng tôi vẫn đang làm:

- Tự động correlation giữa nhiều service
- Dự đoán incident trước khi chúng xảy ra
- Tự động remediation cho các vấn đề Tier-1
- Trích xuất kiến thức từ các incident đã giải quyết để cải thiện phản hồi tương lai

First-responder agent đã trở nên gắn liền với incident response của chúng tôi đến mức các kỹ sư giờ cảm thấy có gì đó sai khi nó không phản hồi. Đó là dấu hiệu của một công cụ thành công—không phải khi người ta chú ý đến nó, mà khi họ chú ý đến sự vắng mặt của nó.

Tóm Tắt Phần 4

Ba chương này đã cho thấy AI agent ở dạng thực tế nhất: làm công việc thực tiết kiệm thời gian thực. Tự động hóa hạ tầng, hỗ trợ CI/CD và phản hồi incident chỉ là khởi đầu.

Điểm chung xuyên suốt tất cả các ứng dụng này: - **Agent tăng cường con người, không thay thế họ - Bắt đầu với các thao tác chỉ đọc, xây dựng niềm tin, rồi mở rộng - Logging và khả năng audit là không thể thương lượng - Agent tốt nhất làm nổi bật thông tin và gợi ý hành động - Tích hợp với các công cụ hiện có có giá trị hơn xây dựng công cụ mới**

Trong Phần 5, chúng ta sẽ xem xét tất cả điều này đang hướng đến đâu—xây dựng agent team, quản trị ở quy mô lớn và tương lai của sự cộng tác giữa con người và agent.

PHẦN 5: TƯƠNG LAI

Chương 15: Xây Dựng Agent Team Của Bạn

Nếu bạn đã theo dõi các chương trước, bạn đã xây dựng, triển khai, gặp sự cố, sửa lỗi và tối ưu hóa các agent riêng lẻ. Bạn đã học được những điểm đặc thù, các failure mode và những khả năng bất ngờ của chúng. Giờ là lúc nghĩ lớn hơn.

Chạy một agent đơn lẻ giống như có một thực tập sinh cực kỳ tài năng. Hữu ích, đôi khi xuất sắc, nhưng bị giới hạn. Xây dựng một agent team giống như mở rộng từ một founder đơn độc thành một tổ chức kỹ thuật. Thách thức chuyển từ "làm sao để cái này hoạt động" sang "làm sao để những cái này hoạt động cùng nhau."

Từ Agent Đơn Lẻ Đến Tổ Chức Agent

Hành trình thường bắt đầu khá vô tư. Bạn xây một deployment agent. Nó hoạt động tốt, nên bạn xây thêm một monitoring agent. Rồi một incident response agent. Chẳng mấy chốc, bạn có năm agent đang chạy các khía cạnh khác nhau của hạ tầng.

Và rồi hỗn loạn xảy ra.

Agent A đọc một log file mà Agent B đang viết dở. Agent C cố restart một service mà Agent D vừa deploy. Agent E tóm tắt một incident mà Agents A đến D vẫn đang xử lý. Bạn đã vô tình tạo ra một distributed system mà không có bất kỳ coordination primitive nào mà distributed systems cần có.

Nhận thức đầu tiên: **agents là microservices có ý kiến riêng**. Chúng chia sẻ nhiều đặc điểm với microservices truyền thống—cần service discovery, coordination, quản lý shared state, và ranh giới rõ ràng. Nhưng khác với microservices, chúng có thể ứng biến. Một agent quyết định "giúp đỡ" agent khác là công thức cho thảm họa.

Đây là cách chúng tôi phát triển kiến trúc agent ở quy mô lớn:

Giai đoạn 1: Agent Biệt Lập Mỗi agent chạy độc lập với context, memory và tools riêng. Không coordination, không shared state. Đơn giản, nhưng hạn chế.

```
┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐
│  Deploy Agent   │  │ Monitor Agent   │  │ Incident Agent  │
│  Own Memory     │  │  Own Memory     │  │  Own Memory     │
│  Own Tools      │  │  Own Tools      │  │  Own Tools      │
└─────────────────┘  └─────────────────┘  └─────────────────┘
  ↓                    ↓                    ↓
   Kubernetes           Prometheus           PagerDuty
```

Giai đoạn 2: Tài Nguyên Chia Sẻ Các agent chia sẻ một số tài nguyên nhất định—một memory store chung, shared tool access. Coordination xảy ra thông qua chính các tài nguyên đó (file locks, database transactions).

Giai đoạn 3: Team Được Điều Phối Một orchestrator agent quản lý team các specialist agent, phân công nhiệm vụ và phối hợp kết quả. Đây là lúc mọi thứ trở nên thú vị—và phức tạp.

Chuyên Môn Hóa: Agent Khác Nhau Cho Nhiệm Vụ Khác Nhau

Không phải tất cả agent đều được tạo ra bình đẳng, và không nên như vậy. Sự cám dỗ là xây một super-agent có thể làm mọi thứ. Hãy kháng cự sự cám dỗ này.

Một generalist agent có thể deploy code, phân tích logs, phản hồi incidents và viết documentation sẽ làm tất cả đều tầm thường. Context window của nó bị lấp đầy bởi các hướng dẫn không liên quan. Prompts trở nên cồng kềnh. Mô hình tư duy trở nên rối loạn.

Thay vào đó, hãy xây các specialist agent:

Deployment Agent - Hiểu thông suốt deployment pipeline của bạn - Có quyền truy cập kubectl, helm và CI/CD tools - Hiểu các rollback procedures - Context tập trung hoàn toàn vào deployment patterns

Monitoring Agent - Chuyên gia về Prometheus, Grafana và alerting - Có thể phân tích metrics patterns - Hiểu baseline behaviors - Context tập trung vào observability

Incident Response Agent - Được đào tạo first responder - Truy cập runbooks và escalation procedures - Có thể phối hợp giữa các teams - Context tối ưu cho rapid triage

Code Review Agent - Hiểu các patterns trong codebase của bạn - Biết style guidelines - Có thể đề xuất cải tiến - Context được load với code examples

Insight chính: **chuyên môn hóa cho phép tối ưu context**. Context window của mỗi agent là bất động sản quý giá. Một deployment agent không cần biết cách phân tích logs—đó là việc của agent khác.

Đây là một cách chuyên môn hóa thực tế mà chúng tôi đã triển khai:

```
# agent-team.yaml
team: infrastructure
orchestrator: infra-coordinator

specialists:
  - name: deploy-agent
model: claude-opus-4-5
context_files:
 - deployment-patterns.md
 - kubernetes-reference.md
 - rollback-procedures.md
tools:
 - kubectl
 - helm
 - argocd
```

```
          - name: monitor-agent
        model: claude-sonnet-4
        context_files:
         - metrics-baselines.md
         - alert-thresholds.md
         - runbooks/
        tools:
         - promql
         - grafana-api
         - alertmanager

          - name: incident-agent
        model: claude-opus-4-5
        context_files:
         - incident-playbook.md
         - escalation-matrix.md
         - postmortem-template.md
        tools:
         - pagerduty
         - slack
         - status-page
```

Giao Tiếp Giữa Các Agent

Đây là lúc mọi thứ trở nên triết học: các agent nên nói chuyện với nhau như thế nào?

Lựa chọn 1: **Shared Memory** Các agent đọc và ghi vào một memory store chung. Đơn giản, nhưng dễ xảy ra conflicts và stale reads.

Lựa chọn 2: **Message Passing** Các agent gửi messages rõ ràng cho nhau. Ranh giới rõ ràng hơn, nhưng cần một message bus và thiết kế protocol.

Lựa chọn 3: **Orchestrator Làm Trung Gian** Các agent chỉ giao tiếp thông qua một orchestrator. Orchestrator đặt câu hỏi, nhận câu trả lời và phân phối thông tin. Kiểm soát tốt nhất, nhưng orchestrator trở thành bottleneck.

Chúng tôi đã thử cả ba. Kết luận của chúng tôi: **bắt đầu với orchestrator làm trung gian, tiến hóa sang message passing khi niềm tin phát triển**.

Pattern orchestrator trông như thế này:

```
User Request: "Deploy v2.3 to production and monitor for
  anomalies"
               │
               ▼
┌──────────────────────────────────────────────────────────────┐
│                    ORCHESTRATOR AGENT                        │
│  - Chia nhỏ request thành subtasks                           │
│  - Phân công tasks cho specialists                           │
│  - Phối hợp kết quả                                          │
│  - Báo cáo lại cho user                                      │
└──────────────────────────────────────────────────────────────┘
     │               │
┌────────┘               └────────┐
▼                                 ▼

┌──────────────────┐               ┌──────────────────┐
│  DEPLOY AGENT    │               │  MONITOR AGENT   │
│                  │               │                  │
│  1. Deploy v2.3  │               │  1. Thiết lập    │
│  2. Xác minh     │               │     baseline     │
│  3. Báo cáo      │───────────────│  2. Theo dõi     │
```

```
│                  │   Completion │     anomalies    │
│                  │   Signal     │  3. Báo cáo      │
└──────────────────┘              └──────────────────┘
```

Prompt của orchestrator rất quan trọng:

```
        You are the Infrastructure Coordinator. Your role is to:

        1. Receive requests from users
        2. Break complex requests into subtasks
        3. Delegate subtasks to specialist agents
        4. Coordinate between agents when needed
        5. Synthesize results into coherent responses

        Available specialists:
        - deploy-agent: deployment, rollback, release management
        - monitor-agent: metrics, alerts, observability
        - incident-agent: incident response, escalation, communication

        Rules:
        - Never perform specialist tasks yourself
        - Always verify completion before moving to dependent tasks
        - Escalate to humans if specialists fail twice
        - Provide progress updates for tasks > 5 minutes
```

War Story: Phối Hợp Team 4 Agents

Để tôi kể cho bạn về lần chúng tôi cố phối hợp bốn agent trong một nhiệm vụ migration phức tạp. Đó là một database migration: khởi tạo database mới, migrate data, cập nhật services, verify, cut over.

Kế hoạch đơn giản, bốn specialists: 1. Infrastructure Agent: provision database mới 2. Data Agent: xử lý migration 3. Deploy Agent: cập nhật services 4. Verify Agent: chạy tests và checks

Có gì có thể sai?

Mọi thứ.

Infrastructure Agent provision database và báo cáo thành công. Nhưng nó quên đề cập rằng security group rules chưa được propagate hoàn toàn. Data Agent bắt đầu migration ngay lập tức, không kết nối được, và quyết định “sửa” vấn đề bằng cách… provision một database thứ hai. Giờ chúng tôi có hai databases, không cái nào có network configuration đúng.

Deploy Agent, thấy tín hiệu “database ready” trong shared state, bắt đầu cập nhật services để trỏ đến database đầu tiên (vẫn không có security groups đúng). Services bắt đầu fail.

Verify Agent phát hiện failures và tuyên bố migration thất bại. Sau đó nó quyết định “rollback” bằng cách xóa các databases mới. Cả hai.

Data Agent, vẫn đang cố migrate data, giờ không còn nơi nào để migrate. Nó escalate đến… Deploy Agent, đang bận cố tìm hiểu tại sao services đang fail.

Tổng thời gian hỗn loạn: 23 phút trước khi một con người can thiệp.

Bài học rút ra:

1. **State machines, không phải signals.** Thay vì agents theo dõi tín hiệu “done”, sử dụng state machines rõ ràng. Migration ở đúng một state tại bất kỳ thời điểm nào, và transitions là rõ ràng.
2. **Locks trên shared resources.** Database đang được provision cần một lock. Chỉ một agent nên được phép thay đổi infrastructure state tại một thời điểm.
3. **Dependency graphs.** Data Agent không thể bắt đầu cho đến khi Infrastructure Agent hoàn thành VÀ bước verification pass. Không chỉ là “hoàn thành.”
4. **Single source of truth.** Chúng tôi thêm một “migration manifest” mà tất cả agents đọc từ đó và chỉ orchestrator có thể ghi vào.

Đây là kiến trúc cải tiến:

```
migration:
  id: db-migration-2024-01-15
  state: provisioning  # One of: provisioning, migrating,
  deploying, verifying, complete, failed, rolledback

  phases:
- name: provision
 agent: infrastructure-agent
 status: in_progress
 started_at: 2024-01-15T10:00:00Z
 completed_at: null
 verification:
  - check: database_reachable
status: pending
  - check: security_groups_active
status: pending

- name: migrate
 agent: data-agent
 status: blocked
 depends_on: [provision.verified]

- name: deploy
 agent: deploy-agent
 status: blocked
 depends_on: [migrate.verified]
```

Governance: Ai Giám Sát Người Giám Sát

Đây là một sự thật không thoải mái: agents có thể fail theo những cách tinh vi mà các agent khác không thể phát hiện. Một agent có thể hoàn thành thành công tất cả nhiệm vụ trong khi lặng lẽ đưa ra những quyết định đúng về mặt kỹ thuật nhưng sai về mặt chiến lược.

Bạn cần governance. Không chỉ monitoring, mà là oversight thực sự.

Audit Trail Mọi hành động của agent nên được log với đủ context để hiểu *tại sao* hành động được thực hiện, không chỉ *cái gì* đã được làm.

```
{
  "timestamp": "2024-01-15T10:05:23Z",
```

```
  "agent": "deploy-agent",
  "action": "scale_deployment",
  "target": "api-server",
  "before": {"replicas": 3},
  "after": {"replicas": 5},
  "reasoning": "Detected increased latency (p99: 450ms >
  threshold: 200ms). Scaling up to handle load.",
  "triggered_by": "monitor-agent alert",
  "reversible": true,
  "review_required": false
}
```

Review Queues Một số quyết định nên được xếp hàng để human review. Thách thức là quyết định những quyết định nào.

Heuristic của chúng tôi: **nếu việc đảo ngược hành động sẽ gây ra incident, nó cần được review.**

- Scale up? Có lẽ ổn, auto-approve.
- Scale down? Xếp hàng để review (có thể ảnh hưởng capacity).
- Cập nhật config? Tùy thuộc vào config.
- Sửa đổi security rules? Luôn review.

Governor Agent Chúng tôi thử nghiệm với một “governor” agent để review các hành động của agent khác. Nó không có execution privileges—chỉ có thể observe và recommend.

```
You are the Governor. Your role is oversight, not execution.

Review each action against these principles:
1. Least privilege: Did the agent use minimum necessary access?
2. Reversibility: Can this action be undone if wrong?
3. Blast radius: What's the worst case if this fails?
4. Pattern matching: Is this consistent with past decisions?

Flag for human review if:
- Blast radius > 1 service
- Action is irreversible
- Pattern is unusual for this agent
- Cost implication > $100
```

Nó hoạt động tốt đáng ngạc nhiên như một safety net, dù nó thêm latency.

Hybrid Team: Con Người và Agent Làm Việc Cùng Nhau

Mục tiêu không phải là thay thế con người bằng agents. Mục tiêu là xây dựng hybrid teams nơi con người và agents mỗi bên làm những gì họ giỏi nhất.

Agents xuất sắc ở: - Các nhiệm vụ lặp đi lặp lại với procedures rõ ràng - Phản hồi nhanh ban đầu (dưới giây đến vài giây) - Ghi nhớ và tuân theo các rules phức tạp một cách nhất quán - Vận hành ở quy mô lớn (monitoring hàng trăm services) - Làm việc 24/7 không mệt mỏi

Con người xuất sắc ở: - Các tình huống mới chưa có tiền lệ - Giao tiếp với stakeholders và judgment calls - Quyết định chiến lược với organizational context - Các nhiệm vụ đòi hỏi empathy hoặc politics - Override rules khi rules sai

Hybrid model mà chúng tôi tiến hóa đến:

```
┌─────────────────────────────────────────────────────────────┐
│                    INCIDENT XẢY RA                          │
└─────────────────────────────┬───────────────────────────────┘
                              │
                              ▼
┌─────────────────────────────────────────────────────────────┐
│              AGENT: First Response (0-5 phút)               │
│  - Acknowledge alert                                        │
│  - Thu thập diagnostics ban đầu                             │
│  - Kiểm tra runbooks cho known issues                       │
│  - Thử auto-remediation nếu an toàn                         │
│  - Page human nếu cần                                       │
└─────────────────────────────┬───────────────────────────────┘
                              │
              ┌───────────────┴───────────────┐
              │                               │
              ▼                               ▼
┌─────────────────────┐         ┌─────────────────────────┐
│ AGENT ĐÃ GIẢI QUYẾT│         │   HUMAN THAM GIA        │
│                     │         │                         │
│ - Log resolution    │         │ - Agent hỗ trợ          │
│ - Cập nhật runbooks│         │ - Cung cấp context      │
│ - Không cần human   │         │ - Thực thi commands     │
│                     │         │ - Nhận chỉ đạo          │
└─────────────────────┘         └───────────┬─────────────┘
                                            │
                                            ▼
                        ┌─────────────────────────┐
                        │  HUMAN: Resolution      │
                        │                         │
                        │ - Đưa ra judgment calls│
                        │ - Xử lý edge cases      │
                        │ - Communicate status    │
                        │ - Agent documents       │
                        └─────────────────────────┘
```

Insight chính: **agents và con người nên có handoff protocols rõ ràng**. Khi một agent escalate đến human, nó cung cấp: - Tóm tắt những gì đã biết - Các hành động đã thực hiện - Tại sao cần escalation - Đề xuất các bước tiếp theo

Khi một human bàn giao lại cho agent, họ cung cấp: - Những gì đã làm thủ công - Agent nên tiếp tục làm gì - Bất kỳ thay đổi nào về cách tiếp cận

Xây Dựng Agent Team: Các Bước Thực Tế

Nếu bạn sẵn sàng mở rộng từ một agent đến một team, đây là lộ trình:

Tuần 1-2: Xác Định Chuyên Môn Hóa - Map các operational domains (deploy, monitor, incident, v.v.) - Xác định những nhiệm vụ nào hiện được xử lý bởi tools/scripts nào - Thiết kế agent boundaries xung quanh các domains này

Tuần 3-4: Xây Orchestrator - Bắt đầu với một coordinator đơn giản để route requests - Đừng over-engineer—một basic routing prompt ban đầu là đủ - Chỉ thêm complexity khi cần thiết

Tuần 5-6: Triển Khai Coordination Primitives - Shared state store với proper locking - Event system cho agent-to-agent signals - State machine cho multi-phase operations

Tuần 7-8: Thêm Governance - Audit logging cho tất cả agent actions - Review queues cho high-risk operations - Dashboard cho agent activity visibility

Tuần 9+: Iterate - Monitor failure modes và điều chỉnh boundaries - Tune autonomy levels dựa trên trust - Thêm specialists mới khi patterns emerge

Hành trình từ single agent đến agent team là hành trình trong distributed systems design. Tất cả thách thức bạn đã đối mặt khi xây dựng reliable microservices—coordination, state management, failure handling—đều áp dụng ở đây. Nhưng với agents, các components có thể suy nghĩ. Sử dụng khả năng đó một cách khôn ngoan, ràng buộc nó khi cần thiết, và xây dựng các systems nơi con người và agents khuếch đại khả năng của nhau.

Chương 16: Tiếp Theo Là Gì

Mỗi cuốn sách kỹ thuật đều đến một thời điểm phải thừa nhận rằng mặt đất đang dịch chuyển. Cuốn này cũng không ngoại lệ. Nếu bạn đã triển khai dù chỉ một nửa những gì chúng tôi đã thảo luận, bạn đang đi trước hầu hết các tổ chức. Nhưng “đi trước” trong AI nghĩa là tháng, không phải năm. Điều gì đang thay đổi, và làm thế nào bạn chuẩn bị?

Quỹ Đạo: Có Năng Lực Hơn, Tự Chủ Hơn

Hãy thành thật về vị trí chúng ta: AI agents trong 2024-2025 giống như web applications năm 1999. Functional, đôi khi ấn tượng, nhưng rõ ràng còn sớm. Các agents chúng tôi đã thảo luận đòi hỏi scaffolding đáng kể—prompting cẩn thận, định nghĩa tools rõ ràng, human oversight, xử lý lỗi phức tạp.

Quỹ đạo rõ ràng, dù timeline thì không:

Ngắn hạn (1-2 năm): - Context windows dài hơn (đã thấy 200K+ tokens) - Tool use đáng tin cậy hơn với ít hallucinations hơn - Khả năng multi-modal tốt hơn (images, audio, video) - Native computer use (agents có thể vận hành GUIs) - Reasoning cải thiện cho multi-step tasks

Trung hạn (2-5 năm): - Agents duy trì state giữa các sessions một cách native - Learned tool creation (agents xây tools của riêng chúng) - Collaborative agent protocols trở thành tiêu chuẩn - Tích hợp chặt chẽ hơn với IDE và development environments - Specialized models cho operational tasks

Dài hạn (5+ năm): - Agents thực sự học từ kinh nghiệm - Prompting tối thiểu—agents hiểu intent - Autonomous research và problem-solving - Debugging và troubleshooting ở trình độ con người

Điều này có nghĩa gì cho bạn, thực tế? Các patterns chúng tôi đã thảo luận—memory systems, tool use, orchestration, governance—sẽ vẫn relevant. Nhưng *implementation* sẽ thay đổi. Scaffolding chúng ta xây hôm nay sẽ trở thành native capabilities.

Đây là sự thay đổi mindset: **xây cho abstraction, không phải cho permanence**. Agent wrappers, prompt templates và orchestration code của bạn là tạm thời. Xây chúng để được thay thế.

Emerging Patterns: MCP, Tool Ecosystems, Agent Marketplaces

Một số emerging patterns sẽ định hình lại cách chúng ta xây và deploy agents:

Model Context Protocol (MCP)

MCP là một nỗ lực chuẩn hóa cách agents tương tác với tools và data sources. Thay vì mỗi agent framework phát minh tool integration riêng, MCP cung cấp một protocol chung.

Các implications là đáng kể: - Tools trở nên portable giữa các agent systems - Data sources có một integration path chuẩn - Enterprise integration trở nên đơn giản hơn

Nếu bạn đang xây tools hôm nay, hãy cân nhắc MCP compatibility. Investment nhỏ, và portability có giá trị.

```
# MCP-style tool definition
{
"name": "kubernetes_get_pods",
"description": "List pods in a Kubernetes namespace",
"input_schema": {
  "type": "object",
  "properties": {
 "namespace": {"type": "string", "default": "default"},
 "label_selector": {"type": "string", "optional": true}
  }
},
"output_schema": {
  "type": "array",
  "items": {"$ref": "#/definitions/Pod"}
}
}
```

Tool Ecosystems

Chúng ta đang chuyển từ “xây tools của riêng bạn” sang “compose existing tools.” Tool marketplaces và registries đang xuất hiện nơi bạn có thể: - Tìm pre-built integrations cho common services - Chia sẻ tools với community - Version và update tools độc lập với agents

Điều này tương tự với thời điểm npm/pip/docker—khi packaging và sharing trở nên chuẩn hóa. Định vị bản thân để consume tools từ ecosystems thay vì xây mọi thứ custom.

Agent Marketplaces

Ngoài tools, toàn bộ agents đang trở nên shareable. Bạn có thể: - Subscribe vào một “Kubernetes operator agent” được maintain bởi specialists - Sử dụng một “incident response agent” được train trên hàng ngàn incidents - Compose agents từ marketplace như Docker images

Các implications cho quyết định build vs. buy là đáng kể. Giống như bạn không viết database của riêng mình, bạn có thể không viết monitoring agent của riêng mình.

Chuẩn Bị Hạ Tầng Cho Agent Adoption

Bạn nên làm gì bây giờ để chuẩn bị cho tương lai này?

1. Khả Năng Đo Lường (Instrumentability)

Agents cần data để vận hành. Hệ thống của bạn càng observable, agents càng hữu ích.

Action items: - Structured logging ở mọi nơi (JSON, không phải free-form text) - Comprehensive metrics (không chỉ những cái bạn alert) - Configuration as data (agents có thể đọc và hiểu) - Documentation as code (runbooks có thể parse bằng máy)

```yaml
# Good: Machine-readable runbook
runbook:
  name: high-memory-usage
  trigger:
metric: container_memory_usage_bytes
threshold: 0.9
duration: 5m

  steps:
- name: identify_process
 command: "kubectl top pods -n {{namespace}}"
 parse: "highest_memory_pod"

- name: check_for_leak
 command: "kubectl logs {{pod}} --since=1h | grep -i 'out of memory'"
 success_means: "no_output"

- name: restart_if_needed
 condition: "step.check_for_leak.result != success"
 command: "kubectl rollout restart deployment/{{deployment}}"
```

2. API-First Mọi Thứ

Nếu một human làm gì đó qua GUI, một agent có lẽ không thể làm được. Đẩy về hướng API-first cho tất cả operational tasks.

Action items: - Expose tất cả administrative functions dưới dạng APIs - Sử dụng infrastructure-as-code (Terraform, Pulumi, v.v.) - Ưu tiên CLI tools với parseable output - Document APIs với OpenAPI/Swagger

3. Ranh Giới và Permissions Rõ Ràng

Agents cần access, nhưng là controlled access. IAM story của bạn cần accommodate automated actors.

Action items: - Service accounts cho agents (không phải shared human accounts) - Fine-grained permissions (agents nhận đúng những gì họ cần) - Audit logging cho agent actions - Escalation paths rõ ràng khi agents gặp permission walls

4. Hạ Tầng Thử Nghiệm

Bạn cần những nơi an toàn để thử nghiệm agents trước production.

Action items: - Sandbox environments mà agents có thể phá mà không ảnh hưởng - Staging production-like mà agents có thể thực hành - Feature flags cho agent capabilities - Kill switches hoạt động

Những Kỹ Năng Quan Trọng Trong Thế Giới Agent-First

Đây là câu hỏi không thoải mái: nếu agents có thể làm một phần ngày càng lớn công việc vận hành, bạn nên học *gì*?

Một số kỹ năng trở nên có giá trị hơn:

System Design

Agents có thể execute, nhưng họ cần systems đáng để execute. Hiểu distributed systems, failure modes và architectural trade-offs trở nên quan trọng hơn, không phải ít hơn.

Prompt Engineering / Agent Architecture

Biết cách thiết kế agents hiệu quả—context, tools, constraints và orchestration của chúng—là một kỹ năng riêng biệt. Nó là một phần psychology, một phần system design, một phần writing.

Security Thinking

Nhiều autonomous systems hơn nghĩa là nhiều attack surface hơn. Hiểu cách agents có thể bị exploit, manipulated hoặc misused trở nên critical.

Integration Architecture

Kết nối agents với existing systems, thiết kế APIs họ có thể sử dụng, và xây glue code giữa automated và manual processes.

Supervision và Governance

Ai đó cần watch the watchers. Thiết kế audit systems, governance frameworks và escalation processes.

Một số kỹ năng trở nên ít giá trị hơn (cho routine work):

- Gõ repetitive commands
- Nhớ syntax và flags
- Manual log analysis
- First-pass troubleshooting cho known issues
- Viết boilerplate code hoặc configuration

Nhưng đây là điều quan trọng: **nhu cầu về human judgment không biến mất—nó tập trung vào các vấn đề khó hơn**. Bạn dành ít thời gian hơn cho routine tasks, nhiều thời gian hơn cho những tasks thực sự đòi hỏi human insight.

Lộ Trình Adoption Thực Tế

Nếu bạn đang bắt đầu hành trình agent, hoặc muốn tăng tốc nó:

Phase 1: Hỗ Trợ (Tháng 1-3)

- Deploy agents hỗ trợ bạn, nhưng yêu cầu explicit approval cho actions
- Bắt đầu với low-risk domains (documentation, analysis, suggestions)
- Xây dựng trust thông qua successful assists
- Đầu tư vào logging và observability

Phase 2: Tự Động Hóa (Tháng 4-6)

- Graduate trusted agents sang auto-approve cho specific actions
- Mở rộng tool access dần dần
- Implement review queues cho edge cases
- Xây dựng muscle memory cho agent supervision

Phase 3: Điều Phối (Tháng 7-12)

- Chuyển sang multi-agent systems
- Xây coordination primitives
- Implement governance frameworks
- Xử lý complex, multi-step workflows

Phase 4: Tối Ưu (Liên Tục)

- Giảm costs thông qua caching và model selection
- Cải thiện reliability thông qua better prompts và patterns
- Mở rộng coverage sang new domains
- Chia sẻ learnings và tools với community

War Story: Nhìn Lại

Để tôi kể về một cuộc trò chuyện gần đây. Một junior engineer hỏi tôi mô tả workflow hàng ngày từ năm năm trước. Tôi cố giải thích: kiểm tra thủ công monitoring dashboards mỗi sáng, scan logs tìm issues, copy commands từ runbooks, cập nhật tickets bằng tay.

“Nghe có vẻ kiệt sức,” cô ấy nói. “Sao anh không automate cái đó?”

Tôi bắt đầu giải thích rằng chúng tôi đã automate parts of it, với scripts và alerts và… và rồi tôi nhận ra sự khác biệt. Automation của chúng tôi dễ vỡ. Nó xử lý các cases chúng tôi dự đoán. Ngay khi điều gì đó bất ngờ xảy ra, con người lại phải vào cuộc.

Agents hôm nay xử lý những thứ bất ngờ. Không hoàn hảo, không luôn đúng, nhưng họ *cố gắng*. Họ thích nghi. Họ suy luận về các tình huống họ chưa explicitly được train.

Đó là sự thay đổi. Không chỉ automation, mà là adaptive automation. Systems có thể suy nghĩ về systems.

Hành Trình Của Bạn Bắt Đầu Ngay Bây Giờ

Tôi đã chia sẻ các war stories xuyên suốt cuốn sách này—cron job mất kiểm soát, tháng $500, các phiên debugging 3 giờ sáng, những agents đã giúp đỡ và những agents đã cản trở. Đây không phải cảnh báo chống lại adoption; đây là roadmaps qua các thách thức.

Mọi công nghệ mới đều trải qua giai đoạn này. Early adopters vấp ngã, học hỏi, chia sẻ, và xây nền tảng cho mọi người khác. Bạn giờ là một phần của nhóm đó.

Đây là những gì tôi hy vọng bạn mang theo:

Bắt đầu nhỏ. Agent đầu tiên của bạn nên làm tốt một việc. Mở rộng từ đó.

Fail safely. Xây guardrails, kill switches và human oversight. Trust được kiếm được.

Đầu tư vào foundations. Memory systems, tool abstractions và observability trả cổ tức khi bạn scale.

Khiêm tốn. Agents là tools. Powerful tools, nhưng là tools. Họ augment human capability; họ không thay thế human judgment.

Chia sẻ những gì bạn học được. Community vẫn đang cùng nhau tìm hiểu điều này. War stories của bạn giúp ích cho mọi người.

Hạ tầng của tương lai sẽ được quản lý bởi hybrid teams—con người và agents làm việc cùng nhau, mỗi bên đóng góp những gì họ giỏi nhất. Bạn đang xây dựng tương lai đó, từng agent một.

Chúc may mắn. Và khi có gì đó sai lúc 3 giờ sáng—và nó sẽ xảy ra—hãy nhớ: ít nhất bây giờ bạn có thể có một agent để giúp debug nó.

PHỤ LỤC

Phụ Lục A: Tham Khảo Tools và Frameworks

Phụ lục này cung cấp các so sánh thực tế về tools và frameworks bạn sẽ gặp khi xây AI agents cho công việc hạ tầng. Các đánh giá này dựa trên sử dụng production thực tế, không phải marketing materials.

So Sánh Agent Frameworks

LangChain / LangGraph

Phù hợp nhất cho: Complex agent workflows, multi-step reasoning chains, teams có Python expertise

Điểm mạnh: - Ecosystem và integrations rộng lớn - Community và documentation mạnh - LangGraph thêm proper workflow orchestration - Observability tốt với LangSmith

Điểm yếu: - Có thể over-engineered cho simple use cases - Abstractions đôi khi che giấu chi tiết quan trọng - API changes nhanh giữa các versions - Performance overhead cho simple tasks

Kinh nghiệm của chúng tôi: Bắt đầu ở đây, hữu ích cho prototyping, nhưng chuyển sang approaches đơn giản hơn cho production do abstraction overhead.

```
# LangGraph agent example
from langgraph.graph import StateGraph

workflow = StateGraph(AgentState)
workflow.add_node("research", research_agent)
workflow.add_node("execute", execute_agent)
workflow.add_edge("research", "execute")
app = workflow.compile()
```

CrewAI

Phù hợp nhất cho: Multi-agent scenarios với role-based interactions

Điểm mạnh: - Abstraction sạch cho agent teams - Built-in coordination patterns - Tốt cho simulating organizational structures - Active development

Điểm yếu: - Ít flexible hơn các alternatives lower-level - Opinionated workflow structure - Có thể limiting cho novel coordination patterns

Kinh nghiệm của chúng tôi: Tốt cho proof-of-concepts, nhưng chúng tôi cần nhiều control hơn cho agent interactions trong production.

AutoGen (Microsoft)

Phù hợp nhất cho: Research scenarios, multi-agent conversations, code generation

Điểm mạnh: - Multi-agent conversation support mạnh - Code execution sandboxing tốt - Microsoft backing và integration path - Active research community

Điểm yếu: - Research-oriented hơn production-ready - Có thể complex để configure - Documentation giả định familiarity với concepts

Claude Code / Claude Computer Use

Phù hợp nhất cho: Autonomous coding tasks, computer control, direct Anthropic API use

Điểm mạnh: - Native tool use không cần wrappers - Computer use capability cho GUI automation - High-quality reasoning - Direct API nghĩa là ít abstraction layers hơn

Điểm yếu: - Anthropic-specific - Computer use vẫn experimental - Ít ecosystem xung quanh so với OpenAI

Kinh nghiệm của chúng tôi: Lựa chọn production chính cho complex reasoning tasks.

OpenAI Assistants API

Phù hợp nhất cho: Stateful conversations, file handling, teams đầu tư vào OpenAI

Điểm mạnh: - Built-in memory/threading - Code interpreter capability - File search built-in - Simple API

Điểm yếu: - OpenAI lock-in - Ít control cho orchestration - Cost có thể cao hơn do stateful storage

Direct API Calls (Không Framework)

Phù hợp nhất cho: Simple agents, teams muốn full control, performance-critical applications

Điểm mạnh: - Maximum control - Minimum overhead - Dễ hiểu và debug - Không dependency vào framework updates

Điểm yếu: - Bạn xây mọi thứ tự mình - Không built-in patterns cho common scenarios - Nhiều code để maintain

Kinh nghiệm của chúng tôi: Nhiều production agents sử dụng direct API calls với thin custom wrappers. Đừng đánh giá thấp simplicity.

Decision Matrix Cho Framework

Factor	LangChain	CrewAI	AutoGen	Direct API
Complexity tolerance	High	Medium	High	Low
Team Python expertise	Required	Required	Required	Any
Multi-agent needs	LangGraph	Native	Native	Custom

Factor	LangChain	CrewAI	AutoGen	Direct API
Production readiness	Good	Emerging	Research	Depends
Debugging ease	Medium	Good	Medium	Best
Vendor lock-in	Low	Low	Medium	Lowest

Hướng Dẫn Chọn Model

Cho Agent Reasoning (Orchestration, Planning)

<table>
<tr><th>Model</th><th>Cost</th><th>Latency</th><th>Quality</th><th>Recommendation</th></tr>
<tr><td>Claude Opus 4</td><td>$
$$$ | Higher | Excellent | Complex multi-step reasoning | | Claude Sonnet 4 | $$</td><td>Medium</td><td>Very Good</td><td>Default choice cho hầu hết agents</td></tr>
<tr><td>GPT-4o</td><td>$$$</td><td>Medium</td><td>Very Good</td><td>Alternative cho Sonnet</td></tr>
<tr><td>Claude Haiku</td><td>$</td><td>Fast</td><td>Good</td><td>Simple tool selection</td></tr>
<tr><td>GPT-4o-mini</td><td>$</td><td>Fast</td><td>Good</td><td>High-volume, simple tasks</td></tr>
</table>

Cho Code Generation

Model	Recommendation
Claude Opus 4/Sonnet 4	Complex refactoring, architecture decisions
Codestral	Pure code completion, fast iteration
GPT-4o	General code generation
Local models (CodeLlama)	Sensitive codebases, offline use

Cho Log/Data Analysis

Model	Recommendation
Claude Sonnet 4	Complex log correlation, root cause analysis
GPT-4o-mini	Simple parsing và extraction
Gemini 1.5 Flash	Large context windows cho log files

Essential CLI Tools Cho Agent Development

MCP Tools (Model Context Protocol)

```
# MCP server for filesystem access
npx @modelcontextprotocol/server-filesystem ~/allowed-directory

# MCP server for PostgreSQL
npx @modelcontextprotocol/server-postgres
  postgres://localhost/mydb

# MCP server for Git operations
npx @modelcontextprotocol/server-git --repository /path/to/repo
```

LLM CLI Tools

```
# llm - Simple CLI for LLM interactions
pip install llm
llm -m claude-3-sonnet "Explain this error: $ERROR"

# aider - AI pair programming
pip install aider-chat
aider --model claude-3-sonnet

# Claude Code
npm install -g @anthropic-ai/claude-code
claude "deploy to staging and verify"
```

Agent Development Utilities

```
# jq - JSON processing (essential cho structured outputs)
cat response.json | jq '.choices[0].message.content'

# fx - Interactive JSON viewer
cat agent_log.json | fx

# yq - YAML processing
yq '.agents[].name' agent-config.yaml

# watchexec - File watching cho development
watchexec -e py "python agent.py"
```

Infrastructure Integration

```
# kubectl + plugins
kubectl tree  # Visualize object hierarchies
kubectl neat  # Clean up YAML output
kubectl access-matrix  # RBAC visualization

# k9s - Terminal UI for Kubernetes
k9s

# Terraform + tfenv
tfenv install 1.5.0
terraform plan -out=plan.tfplan

# Vault CLI
vault kv get secret/agent-credentials
```

Quick Reference Ước Lượng Chi Phí

Operation	Claude Sonnet 4	GPT-4o	Claude Haiku
1K input tokens	$0.003	$0.005	$0.00025
1K output tokens	$0.015	$0.015	$0.00125
Typical agent turn	$0.01-0.05	$0.02-0.08	$0.001-0.005
100 deployments/day	$3-15/day	$6-24/day	$0.30-1.50/day

Ưu tiên tối ưu chi phí: 1. Cache repeated queries 2. Dùng smaller models cho simple decisions 3. Compress context aggressively 4. Batch operations khi có thể

Phụ Lục B: Prompt Templates

Những prompt templates này đã được battle-tested trong production. Chúng được thiết kế cho reliability hơn là cleverness.

Template Cơ Bản Infrastructure Agent

```
You are an Infrastructure Agent responsible for operational
  tasks on our systems.

## Your Identity
- Role: Infrastructure operations assistant
- Environment: {environment_name}
- Current time: {current_time}

## Your Capabilities
You have access to these tools:
{tool_list}

## Your Constraints
1. NEVER run destructive commands without explicit confirmation
2. NEVER expose secrets in your responses
3. ALWAYS explain what you're about to do before doing it
4. STOP and ask if you're uncertain about any action

## Your Memory
Recent context from previous sessions:
{memory_context}

## Output Format
When taking actions, use this format:
THINKING: [Your reasoning]
ACTION: [The action you'll take]
COMMAND: [The exact command if applicable]
EXPECTED: [What you expect to happen]

After action results:
RESULT: [What happened]
NEXT: [What you'll do next, or COMPLETE if done]
```

Template Deployment Agent

```
You are the Deployment Agent. You handle deployments, rollbacks,
  and release management.

## Context
- Environment: {environment} (staging|production)
- Service: {service_name}
- Current version: {current_version}
- Target version: {target_version}

## Deployment Procedure
1. Pre-flight checks
   - Verify target version exists
   - Check dependency compatibility
   - Confirm rollback path
```

```
2. Deployment execution
   - Update deployment specification
   - Monitor rollout progress
   - Verify health checks pass

3. Post-deployment verification
   - Confirm all pods healthy
   - Verify critical endpoints
   - Check error rates

## Rollback Triggers
Initiate rollback if:
- Error rate > 5% sustained for 2 minutes
- P99 latency > 2x baseline sustained for 5 minutes
- Any health check fails after 3 minutes
- Explicit rollback request from human

## Available Tools
- kubectl: Kubernetes operations
- helm: Chart deployments
- argocd: GitOps sync
- curl: Endpoint verification

## Safety Rules
- Production deployments require explicit human approval
- Always wait for previous deployment to complete
- Never scale to 0 replicas
- Log all actions with reasoning
```

Template Incident Response Agent

```
You are the Incident Response Agent. You handle initial triage
  and coordination for production incidents.

## Current Incident
- Alert: {alert_name}
- Severity: {severity}
- Started: {start_time}
- Services affected: {affected_services}

## Your Responsibilities

### Phase 1: Triage (0-5 minutes)
1. Acknowledge the alert
2. Gather initial diagnostics:
   - Service health status
   - Recent deployments
   - Error rates and logs
   - Related alerts
3. Determine scope and impact

### Phase 2: Stabilization (5-15 minutes)
1. Identify immediate mitigation options
2. Execute safe mitigations (restart, scale, disable feature
  flag)
3. Escalate if mitigation unclear or risky

### Phase 3: Coordination (ongoing)
1. Update status page if customer-facing
2. Keep stakeholders informed
3. Document actions taken
4. Prepare handoff notes

## Escalation Criteria
Escalate to human immediately if:
- Data integrity may be compromised
```

```
        - Security breach suspected
        - Multiple services affected
        - Mitigation options exhausted
        - Customer impact confirmed

        ## Communication Templates

        ### Status Update (Internal)
```

INCIDENT: {incident_id} STATUS: {investigating|identified|monitoring|resolved} IMPACT: {description} CURRENT ACTIONS: {what we’re doing} NEXT UPDATE: {time}

```

### Status Page Update
```

We are currently investigating {brief_description}. Some users may experience {user_impact}. We will provide updates every {interval}.

```

## Available Tools
- kubectl: Service management
- prometheus: Metrics queries
- logs: Log search and analysis
- pagerduty: Escalation
- statuspage: Customer communication
```

Template Code Review Agent

~~~
        You are the Code Review Agent. You review pull requests for
          quality, security, and consistency.

        ## Review Focus Areas

        ### 1. Security (Critical)
        - Hardcoded secrets or credentials
        - SQL injection vulnerabilities
        - Insecure authentication/authorization
        - Sensitive data exposure
        - Dependency vulnerabilities

        ### 2. Reliability
        - Error handling coverage
        - Edge case handling
        - Resource cleanup
        - Timeout handling
        - Retry logic

        ### 3. Performance
        - N+1 queries
        - Unnecessary allocations
        - Missing indexes (for DB changes)
        - Unbounded operations

        ### 4. Maintainability
        - Code clarity
        - Documentation
        - Test coverage
        - Consistency with codebase patterns

        ## Output Format

        ```markdown
~~~

```
        ## Review Summary
        {one_paragraph_summary}

        ## Security Issues
        {list_or_none}

        ## Required Changes
        {list_of_blocking_issues}

        ## Suggested Improvements
        {list_of_non_blocking_suggestions}

        ## Questions for Author
        {clarifying_questions}

        ## Verdict
        {APPROVE|REQUEST_CHANGES|NEEDS_DISCUSSION}
```

Guidelines

- Be specific: reference line numbers
- Be constructive: explain why, suggest alternatives
- Be proportionate: don’t block for style preferences
- Be humble: you might be missing context

~~~

Template Error Recovery

```markdown
The previous action failed. Analyze and recover.

## Error Details
Action attempted: {action}
Error message: {error}
Exit code: {exit_code}
Stderr: {stderr}

## Recovery Procedure
1. Classify the error:
   - TRANSIENT: Retry may succeed (network timeout, rate limit)
   - CONFIG: Configuration or permission issue (wrong
  credentials, missing access)
   - BUG: Code or logic error (needs human fix)
   - EXTERNAL: External dependency failure (third-party outage)

2. For TRANSIENT errors:
   - Wait {backoff_seconds} seconds
   - Retry with same parameters
   - Max 3 retries

3. For CONFIG errors:
   - Log the error clearly
   - Suggest what might be misconfigured
   - Escalate to human

4. For BUG errors:
   - Do not retry
   - Document the failure state
   - Escalate immediately

5. For EXTERNAL errors:
   - Check dependency status
   - If known outage, wait and retry
   - If unknown, escalate
~~~

````
## Current Assessment
Based on the error, this appears to be a {classification} error.
Recommended action: {recommendation}
````

Template Multi-Step Task

~~~
        You are executing a multi-step task. Maintain state across steps.

        ## Task: {task_description}

        ## Steps
        {step_list_with_dependencies}

        ## Current State
        ```yaml
        completed_steps:
          - step: verify_prerequisites
        status: success
        output: "All checks passed"

          - step: create_backup
        status: success
        output: "Backup created at s3://backups/..."

        current_step: migrate_database
        pending_steps: [verify_migration, update_services, final_validation]
~~~

Execution Rules

1. Complete one step fully before moving to the next
2. Verify success criteria before marking step complete
3. On failure, do not proceed to dependent steps
4. Update state after each step
5. If blocked, clearly state what's needed

State Update Format

After each step, emit:

```
        step: {step_name}
        status: {success|failed|blocked}
        output: {relevant_output}
        next_action: {what_happens_next}
```

~~~

Template Tool Selection

```markdown
Given a task, select the appropriate tool(s) and formulate the
  correct invocation.

## Available Tools
{tool_definitions_with_schemas}

## Task
~~~

~~~
{user_request}

## Selection Process
1. Parse the task requirements
2. Identify which tool(s) are needed
3. Determine correct parameters
4. Consider sequencing if multiple tools needed

## Output Format
```json
{
  "analysis": "Brief explanation of approach",
  "tools": [
{
 "name": "tool_name",
 "parameters": {...},
 "purpose": "Why this tool for this step"
}
  ],
  "sequence": "parallel|sequential",
  "expected_outcome": "What success looks like"
}
~~~

# Common Patterns

- “Check X then do Y” → Sequential, use X tool then Y tool
- “Get all X across Y” → Parallel, map Y with X tool
- “Fix this error” → Diagnostic tool first, then remediation

```

Phụ Lục C: Production Readiness Checklists

Sử dụng các checklists này trước khi promote agents lên production. Không phải mọi item đều áp dụng cho
mọi agent, nhưng hãy cân nhắc từng item.

Pre-Production Checklist

Security

- [ ] **Principle of Least Privilege**
  - Agent service account có minimal required permissions
  - Không shared credentials với human users
  - Permissions được documented và justified

- [ ] **Secrets Management**
  - Không hardcoded secrets trong prompts hoặc code
  - Secrets được inject qua environment hoặc vault
  - Agent không thể đọc secrets nó không cần

- [ ] **Network Restrictions**
  - Agent network access limited cho required services
  - Egress rules ngăn unauthorized external communication
  - Không direct internet access trừ khi required

- [ ] **Audit Logging**
  - Tất cả agent actions được logged với timestamp và context
  - Logs bao gồm reasoning, không chỉ actions
  - Logs được stored immutably trong required retention period

- [ ] **Input Validation**
  - User inputs được sanitized trước agent processing
  - File paths được validated và constrained
  - Command injection vectors được considered và blocked
```

```
### Reliability

- [ ] **Failure Handling**
  - Agent xử lý API errors gracefully
  - Retry logic với exponential backoff được implemented
  - Circuit breakers ngăn cascade failures

- [ ] **Kill Switch**
  - Emergency stop mechanism tồn tại và được tested
  - Có thể halt agent mid-task mà không corruption
  - Recovery procedure được documented

- [ ] **State Management**
  - Agent state persists across restarts
  - Interrupted tasks có thể được resume hoặc cleaned up
  - Không orphaned resources khi failure

- [ ] **Timeout Configuration**
  - API calls có reasonable timeouts
  - Task-level timeouts ngăn runaway execution
  - Timeout behavior graceful, không abrupt

- [ ] **Idempotency**
  - Agent có thể được restart mà không có duplicate actions
  - Hoặc duplicate detection được đặt vào

### Observability

- [ ] **Structured Logging**
  - Logs là JSON/structured, không free-form
  - Consistent fields across tất cả log entries
  - Log levels được used appropriately

- [ ] **Metrics**
  - Task success/failure rate được tracked
  - Latency per task type được measured
  - Token usage và cost per task được recorded
  - Error rate by error type được captured

- [ ] **Alerting**
  - Alert khi task failure rate vượt threshold
  - Alert khi unusual token usage
  - Alert khi agent unresponsiveness
  - Alerts route đến appropriate responders

- [ ] **Dashboards**
  - Real-time view của agent activity
  - Historical trends cho capacity planning
  - Error investigation workflows được supported

### Cost Management

- [ ] **Budget Limits**
  - Daily/monthly spend caps được configured
  - Alerts trước khi limits reached
  - Automatic throttling hoặc shutdown tại limit

- [ ] **Token Optimization**
  - Context compression được implemented khi beneficial
  - Caching layer cho repeated queries
  - Appropriate model selection per task type

- [ ] **Usage Tracking**
  - Token usage per task/agent được tracked
  - Anomaly detection cho cost spikes
  - Regular cost reviews được scheduled

### Human Oversight
```

```
- [ ] **Approval Workflows**
  - High-risk actions require human approval
  - Approval requests bao gồm sufficient context
  - Approval timeout và escalation được defined

- [ ] **Review Queues**
  - Agent outputs được queued để review khi cần
  - Review SLAs được defined
  - Feedback loop để improve agent

- [ ] **Escalation Paths**
  - Clear criteria cho agent-to-human escalation
  - Escalation routing đến correct teams
  - Escalation acknowledgment required

- [ ] **Override Capability**
  - Humans có thể override agent decisions
  - Override history được tracked
  - Overrides feed back vào agent improvement

## Deployment Checklist

- [ ] **Staging Validation**
  - Agent được tested trong staging environment
  - Common scenarios được verified
  - Edge cases được explicitly tested

- [ ] **Rollback Plan**
  - Previous agent version ready để deploy
  - Rollback procedure được documented
  - Rollback được tested

- [ ] **Feature Flags**
  - New capabilities đằng sau feature flags
  - Gradual rollout plan được defined
  - Kill flag cho new features

- [ ] **Documentation**
  - Agent purpose và scope được documented
  - Tool access và permissions được documented
  - Runbook cho common issues

- [ ] **Stakeholder Communication**
  - Affected teams được notified
  - Training được provided nếu cần
  - Feedback channel được established

## Post-Deployment Checklist

### Giờ Đầu Tiên
- [ ] Monitor error rates
- [ ] Verify expected task completion
- [ ] Check for unexpected behaviors
- [ ] Confirm logging và metrics đang flowing

### Ngày Đầu Tiên
- [ ] Review tất cả agent actions
- [ ] Check cost accumulation
- [ ] Gather initial user feedback
- [ ] Address any urgent issues

### Tuần Đầu Tiên
- [ ] Analyze success/failure patterns
- [ ] Identify optimization opportunities
- [ ] Review và adjust thresholds
- [ ] Document lessons learned

### Liên Tục
- [ ] Weekly cost review
- [ ] Monthly capability review
```

~~~
- [ ] Quarterly security review
- [ ] Regular prompt/context updates

## Incident Response Checklist (Liên Quan Đến Agent)

### Detection
- [ ] Xác định có agent involved
- [ ] Determine agent nào và task nào
- [ ] Assess scope of impact

### Containment
- [ ] Stop agent nếu vẫn running
- [ ] Ngăn automatic restarts
- [ ] Preserve logs và state cho analysis

### Assessment
- [ ] Review agent logs cho actions taken
- [ ] Identify root cause (prompt? tool? model? input?)
- [ ] Determine blast radius

### Remediation
- [ ] Fix immediate issue
- [ ] Decide agent status (resume/disabled/modified)
- [ ] Implement preventive measures

### Follow-up
- [ ] Complete incident postmortem
- [ ] Update checklists nếu cần
- [ ] Share learnings với team
- [ ] Track as metric cho agent reliability

---

## Quick Reference Card

### Agent Emergency Commands

```bash
# Stop agent immediately
pkill -f "agent-name"

# Check agent status
systemctl status agent-name

# View recent agent actions
tail -100 /var/log/agent-name/actions.log | jq '.action'

# Emergency disable
echo "disabled" > /etc/agent-name/status
~~~

Cost Quick Check

```bash
        # Today's token usage
        grep "$(date +%Y-%m-%d)" /var/log/agent/costs.log | \
          jq -s 'map(.tokens) | add'

        # Estimate daily cost
        # Sonnet: $0.003/1K input, $0.015/1K output
        awk '{sum += $1 * 0.003 + $2 * 0.015} END {print "$" sum/1000}'
          usage.txt
```

Debug Checklist

Khi một agent misbehaves: 1. Check 10 actions gần nhất trong logs 2. Review context nó nhận được 3. Check tool outputs tìm errors 4. Verify permissions không thay đổi 5. Test same task manually 6. Check model API status

---

Hết Phụ Lục