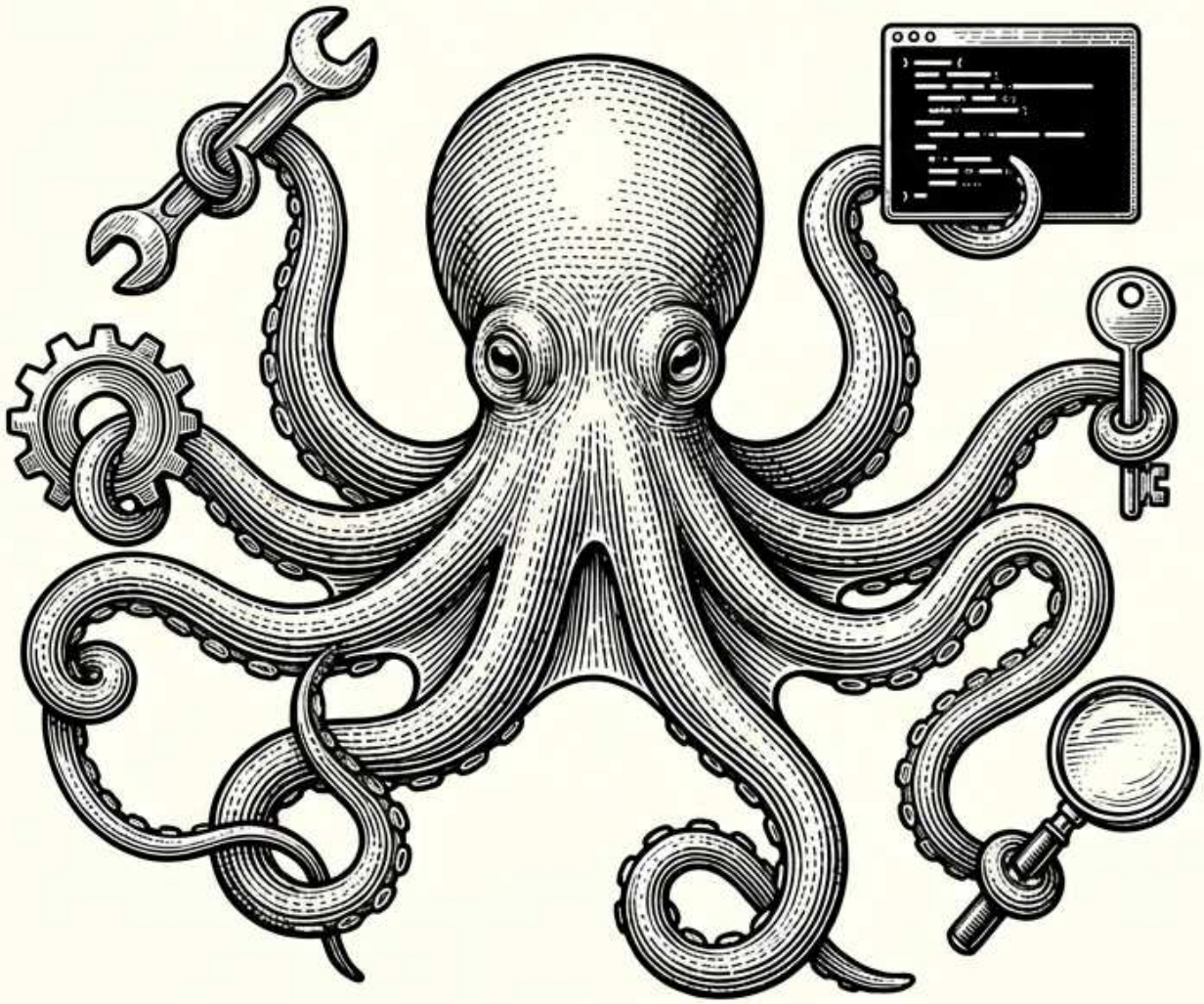
AI Agents in Production

War Stories from a DevOps Engineer



Do Cao Hieu

AI Agents in Production

War Stories from a DevOps Engineer

A practical guide to running AI agents in production environments.

From the lessons learned building and operating autonomous AI systems for infrastructure automation, CI/CD, incident response, and beyond.

PART 1: FOUNDATIONS

Chapter 1: The Shift from Prompts to Agents

The Evolution of Operational Tooling

Every decade or so, the way we manage infrastructure undergoes a fundamental shift. In the 1990s, we typed commands into terminals and hoped we remembered the right flags. In the 2000s, we wrapped those commands into bash scripts—fragile, but at least repeatable. The 2010s brought us Infrastructure as Code: Terraform, Ansible, Puppet. We stopped describing *how* and started describing *what*.

Then came the prompts.

Starting around 2023, a new pattern emerged. Instead of writing scripts, engineers started asking AI to write them. Instead of debugging configuration files line by line, we pasted error messages into ChatGPT and got solutions back. This was revolutionary—but it was also just the beginning.

The progression looks like this:

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

Each step represented a leap in abstraction. CLI commands required knowing exact syntax. Scripts captured that syntax in reusable form. IaC let us declare desired state. Prompts let us express intent in natural language.

But agents? Agents close the loop entirely.

What Makes an “Agent” Different from a “Chatbot”

I’ve lost count of how many times someone has asked me: “Isn’t this just ChatGPT with extra steps?”

Fair question. Let me be precise about the distinction.

A **chatbot** responds to inputs. You ask a question, you get an answer. The conversation might span multiple turns, but fundamentally, the chatbot is reactive. It waits for you. It answers you. It waits again.

An **agent** pursues goals. You give it an objective, and it figures out how to achieve it. It plans. It takes actions. It observes the results of those actions. It adapts. Crucially, it does this *without waiting for you at every step*.

Here's a concrete example. Imagine you want to deploy a new version of a service:

Chatbot interaction:

```
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...
```

This goes on. You're the loop. You execute commands. You report results. You ask follow-up questions. The bot just answers.

Agent interaction:

```
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.
```

See the difference? The agent drives the conversation. It takes actions autonomously. It observes results and proceeds without asking you what to do next.

This is the fundamental shift: from **human-in-the-loop** to **human-on-the-loop**.

The Agent Loop: Plan → Execute → Observe → Adapt

Every agent, regardless of how sophisticated, follows a basic loop:

```
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
```

This is sometimes called the ReAct pattern (Reasoning + Acting), or simply the agent loop. Let's break it down:

Plan: The agent uses its LLM brain to decide what to do next. This isn't a fixed script—it's dynamic reasoning based on current context.

Execute: The agent takes action. This might mean running a command, calling an API, writing a file, or sending a message. This is where tools come in—we'll cover those in Chapter 2.

Observe: The agent sees what happened. Did the command succeed? What did it output? Are there errors? This observation becomes part of the context for the next planning step.

Adapt: If something unexpected happens, the agent adjusts. Maybe the first approach didn't work. Maybe new information changed the plan. Good agents recover gracefully; bad ones spin in infinite loops.

Here's what this looks like in practice:

```
        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."
```

This loop is deceptively simple. The complexity—and the danger—lies in what happens inside each step. What tools does the agent have access to? How does it decide when it's done? What happens when it makes a mistake?

We'll spend most of this book answering those questions.

Why DevOps is the Perfect Domain for AI Agents

Not every domain is equally suited to agents. Some require too much nuanced judgment. Some have consequences too severe for autonomous action. Some lack the tooling for programmatic interaction.

DevOps hits a sweet spot.

Rich tooling ecosystem: Our entire world is built on CLIs and APIs. Kubernetes, Docker, AWS, Terraform, Ansible—everything is designed for programmatic interaction. Agents can plug into existing tools without needing custom interfaces.

Observable outcomes: When we deploy something, we know if it worked. We have metrics, logs, health checks. Agents can verify their actions objectively.

Recoverable failures: Most DevOps mistakes can be fixed. Rollback a deployment. Restore a backup. Terminate a runaway process. This is important—agents will make mistakes, and we need to survive them.

Repetitive patterns: Much of operations work follows patterns. “Check if X, then do Y, unless Z.” “When the alert fires, run through this checklist.” These patterns are perfect for automation, and agents can learn them.

Clear intent: “Deploy to staging.” “Scale up the database.” “Fix this failing test.” DevOps tasks often have unambiguous success criteria.

This doesn’t mean deploying agents in DevOps is easy—it’s not, and we’ll see plenty of war stories proving that. But the domain is tractable. The tools exist. The feedback loops are tight.

Case Study: A Simple Prompt vs an Agent Solving the Same Problem

Let me show you a real scenario that illustrates the difference.

The situation: At 2 AM, an alert fires. API latency has spiked above SLA. Someone needs to investigate and fix it.

The Prompt Approach

An engineer wakes up, opens their laptop, and starts prompting:

```
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...
```

This continues for 20-30 minutes. The engineer is doing all the work. The LLM is just an advisor.

The Agent Approach

The agent wakes up instead of the engineer (or at least, before escalating to them):

```
[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."
```

The agent did in 51 seconds what took the human 20-30 minutes of back-and-forth. More importantly, the engineer got to stay asleep for the simple case—while still being informed and able to review.

This is the promise of agents in DevOps. Not replacing human judgment, but handling the routine so humans can focus on the exceptional.

The Reality Check

Now, I need to be honest with you. The agent example above is the *ideal* case. In reality:

- The agent might have rolled back something that wasn’t the cause
- The agent might have gotten stuck if the rollback failed
- The agent might have missed a more complex root cause
- The agent might have cost $0.50 in API calls for a problem that would have resolved itself

Agents are not magic. They’re a tool—a powerful one, but one that requires careful design, guardrails, and human oversight. The rest of this book is about how to build agents that work in the real world, with all its messiness and edge cases.

Let’s start by understanding what’s inside an agent.

Chapter 2: Anatomy of an AI Agent

Core Components: LLM Brain, Tools, Memory, Orchestration

If you cracked open an AI agent and looked inside, you'd find four fundamental components:

The LLM Brain: This is the reasoning engine. It takes in context—the goal, the current situation, the history of what's happened—and decides what to do next. The LLM doesn't actually *do* anything except think and generate text. But that text can be instructions for actions.

Tools: These are the agent's hands. While the LLM can only think, tools let it act. A tool might be a bash command executor, an API client, a file writer, or anything else that changes the world. When the LLM decides to "run `kubectl get pods`", a tool actually executes that command.

Memory: This is how the agent maintains context across time. Without memory, every interaction would start from zero. Memory lets the agent remember what it was doing, what it tried, what worked, and what didn't.

Orchestration: This is the glue that ties everything together. The orchestration layer manages the agent loop, handles tool execution, maintains memory, and enforces guardrails. It's the runtime that makes an agent an agent rather than just an LLM.

Let's examine each component in detail.

The Context Window: Your Agent's Working Memory

The context window is perhaps the most important constraint to understand. It's the LLM's short-term memory—everything the model can "see" at once.

Different models have different context limits: - GPT-3.5: 16K tokens - GPT-4o: 128K tokens - Claude 3: 200K tokens - Gemini 1.5: 1M+ tokens

But here's what nobody tells you in the marketing materials: just because a model *can* handle 200K tokens doesn't mean it *should*.

```
        # 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"]
```

In production, context window management is everything. Fill it up, and you pay through the nose. Let it overflow, and the agent loses important information. We'll cover strategies in Chapter 5.

Tool Use: Giving Agents Hands to Interact with the World

Tools are how agents affect reality. A tool is simply a function that the LLM can invoke through a structured format.

Here's a basic tool definition:

```
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"]
  }
}
]
```

When the LLM decides to use a tool, it generates a structured response:

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

Your orchestration layer intercepts this, executes the actual command, and feeds the result back to the 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()
```

The critical insight: **tool design is security design**. Every tool you give an agent is a capability it can use—or misuse. We'll cover this extensively in Chapter 8.

Common tool categories in DevOps agents:

Category	Examples
Shell	run_command, run_script
Files	read_file, write_file, list_directory
Network	http_request, curl
Kubernetes	kubectl, helm
Cloud	aws, gcloud, az
Git	git_status, git_commit, git_push
Communication	send_slack, send_email

Memory Systems: Short-term, Long-term, Episodic

Agents need to remember things at different timescales.

Short-term memory is the conversation itself. It's what the agent has seen in this session. This lives in the context window and is naturally limited by token count.

```
        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 persists across sessions. This is where the agent stores information it might need later—project context, user preferences, learned patterns.

```
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 captures sequences of events—what the agent did in the past, what worked, what failed. This is crucial for learning from experience.

```
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()
```

The `MEMORY.md` pattern is a pragmatic approach I've used successfully: a single Markdown file that the agent reads at the start of each session and updates with important information. It's simple but effective:

```
# 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
```

Agent Architectures: Single Agent vs Multi-Agent Systems

Most agents start simple: one agent, one task. But as tasks grow complex, single agents struggle.

Single Agent Architecture:

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

Pros: Simple, easy to debug, clear accountability. Cons: Limited by single context window, can't parallelize.

Multi-Agent Architecture:

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

Pros: Specialization, parallel execution, larger total context. Cons: Complex coordination, inter-agent communication overhead.

A common pattern is the **orchestrator/worker** model:

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

We'll dive deep into multi-agent patterns in Chapter 7.

Chapter 3: Your First Production Agent

Choosing Your Stack: Frameworks, Models, Infrastructure

Before writing any code, you need to make some choices. Let me give you the lay of the land as of early 2025.

Agent Frameworks:

Framework	Pros	Cons
LangChain	Huge ecosystem, lots of examples	Complex, sometimes over-engineered
AutoGPT	Fully autonomous	Hard to control, expensive
CrewAI	Good multi-agent support	Newer, less battle-tested
Claude Code	Excellent tool use, safe defaults	Anthropic-specific
Custom	Full control	More work

For production DevOps agents, I recommend starting simple—either a minimal custom framework or Claude Code if you're using Anthropic's models. The heavy frameworks add complexity you don't need initially.

Models:

Model	Best For	Cost	Context
GPT-4o	General tasks, good tool use	|128*K*|| *Claude*3*Opus*| *Complexreasoning*, *safety*| $	200K
Claude 3 Sonnet	Good balance, fast	$$	200K
GPT-4o-mini	Simple tasks, cost-sensitive	$	128K

For DevOps, Claude models have an edge due to better instruction-following and more conservative defaults—important when your agent has shell access.

Infrastructure:

Your agent needs to run somewhere. Options: - **Local development**: Your laptop, for testing - **Server/VM**: A dedicated machine for production agents - **Container**: Docker with restricted capabilities - **Serverless**: Lambda/Cloud Functions for triggered agents

My recommendation: start with a dedicated VM or container with explicit resource limits. Serverless gets complicated with long-running agent loops.

The “Hello World” of Agents: A Deployment Assistant

Let’s build something real. A deployment assistant that can: 1. Check the current deployment status 2. Deploy new versions 3. Verify health after deployment 4. Roll back if needed

Here’s a minimal implementation:

```
        #!/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")
```

That's about 100 lines of code for a functional agent. Let's test it:

```
$ 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
```

Environment Setup: API Keys, Permissions, Sandboxing

Before your agent touches production, you need proper setup.

API Keys: Store them securely, never in 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)
```

Permissions: Principle of least privilege. Your agent should have exactly the permissions it needs, no more.

```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: Don't run agents with your personal credentials.

```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"]
```

Run with restrictions:

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

Running Your First Autonomous Task

Here's the moment of truth. Instead of interactive prompting, let's have the agent complete a task autonomously:

```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
```

Your first autonomous deployment. It worked.

What Could Go Wrong (Spoiler: A Lot)

Now for the reality check. That happy path above? It's maybe 60% of real-world runs. Here's what goes wrong:

War Story: The Infinite Rollback Loop

At a fintech startup (not mine, I swear), they built a deployment agent similar to what we just created. It worked great—until one Friday evening.

The agent was tasked with deploying a new version. The deployment succeeded, but the health check failed. Following its instructions, the agent rolled back. Then it noticed the deployment wasn't at the target version, so it deployed again. Health check failed. Rolled back. Deployed. Rolled back.

By the time someone noticed on Monday, the agent had executed 847 deployments over the weekend. Their monitoring dashboards showed a perfect sine wave of deployments and rollbacks.

Lesson learned: Always track state. Did I already try this? How many times? Add termination conditions:

```
        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."
```

War Story: The Overly Helpful Agent

An SRE team gave their agent write access to configuration files. The task was simple: “Update the replica count from 3 to 5.”

The agent found the deployment YAML, updated the replica count… and also “helpfully” updated several other fields it thought were suboptimal. It bumped the CPU limits, changed the liveness probe timing, and modified the environment variables to what it thought were “better defaults.”

The deployment failed spectacularly because the new environment variables pointed to non-existent services.

Lesson learned: Agents should do exactly what you ask, nothing more. Be explicit in your system prompts:

```
        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.
        """
```

War Story: The Token Explosion

A team had an agent that analyzed logs to find patterns. They pointed it at a production log file without checking the size first.

The file was 2GB.

The agent tried to read it into context. The API call failed, but not before the agent made several attempts with different chunk sizes. Total token cost for the failed debugging session: $340.

Lesson learned: Always limit input sizes:

```
        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()
```

What Goes Wrong: Summary

Failure Mode	Symptom	Prevention
Infinite loops	Agent retries forever	Max attempts, backoff, state tracking
Scope creep	Agent does more than asked	Explicit constraints, minimal tools
Resource exhaustion	High costs, timeouts	Input limits, token budgets
Silent failures	Agent says “done” but wasn’t	Verification steps, success criteria
Cascading errors	One mistake leads to many	Atomic operations, checkpoints

These failure modes are why the rest of this book exists. Building agents that work in demos is easy. Building agents that work in production—reliably, safely, cost-effectively—is hard.

In Part 2, we’ll dive into the hard lessons: what happens when agents go wrong, and how to build systems that survive it.

End of Part 1: Foundations

In Part 2, we’ll cover the hard lessons learned from production failures: runaway costs, infinite loops, context engineering, memory systems that actually work, and multi-agent coordination.

Part 2: The Hard Lessons

Chapter 4: When Agents Go Wrong

If you've made it this far, you've probably deployed your first agent and felt that rush of excitement watching it autonomously complete tasks. Now let me tell you about the moments that made me question everything.

War Story: The Runaway Cron Job That Burned $200 in Tokens

It was a Tuesday night. I had deployed what I thought was a clever monitoring agent—it would check our services every 5 minutes, analyze any anomalies, and create tickets for issues it found. Simple enough, right?

What I didn't anticipate was the feedback loop.

The agent noticed our ticket queue growing. It interpreted this as a sign of degraded service quality. So it investigated more thoroughly. Each investigation triggered more API calls to gather context. More context meant longer prompts. Longer prompts meant higher token usage. And because the agent was "concerned" about the growing ticket queue (which it was creating), it increased its investigation frequency.

By the time I woke up at 6 AM, the agent had made 847 API calls to Claude, created 312 tickets (most of them duplicates about "elevated ticket volume"), and burned through $200 in API credits.

The fix was embarrassingly simple:

```
        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
```