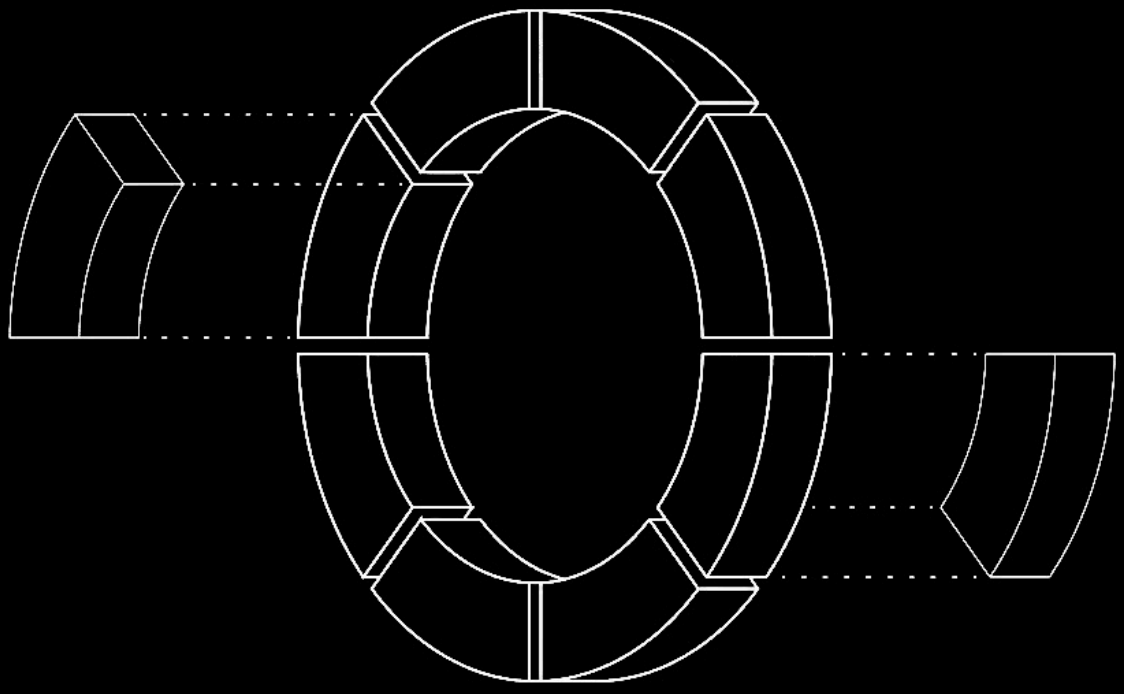
scanner

Building Your First AI SOC Agents: Foundations & Your First Agent



Cliff Crosland
CEO, Co-founder

Why Build SOC Agents Now?

Picture this: It's 2am, an alert fires for unusual database access, and your on-call analyst has three other incidents already queued. They pull up the logs, check the user's history, compare it to baseline behavior, look for related activity. Thirty minutes later, it's a false positive. A scheduled backup job. Meanwhile, two more alerts arrived.

This is the reality for most security teams. Alert volumes grow faster than headcount. Detection rules multiply. Threat intelligence arrives around the clock. Many organizations staff analysts 24/7 to keep up, but manual alert triage is a heavy burden that produces relatively low value. Most alerts are false positives. Investigating each one is repetitive, time-consuming work that keeps your best people focused on routine tasks instead of high-impact projects.

The result is an opportunity cost. Every hour an analyst spends triaging benign alerts is an hour not spent on detection engineering: building new rules to find threats you're currently missing, or tuning existing rules to reduce noise and false positives. When your team is underwater with alert volume, there's no bandwidth left for the strategic work that actually improves your security posture

Autonomous SOC agents change this equation by handling routine triage automatically. When an agent can investigate the bulk of alerts that turn out to be benign, your analysts can focus on the ones that actually matter. More importantly, with triage off their plate, they finally have bandwidth for the work that compounds: building new detections to find threats you're currently missing, tuning existing rules to reduce noise, running threat hunts, and improving your security architecture. The investigation that takes an analyst 30-45 minutes (pull logs, check history, assess baseline, document findings) happens in seconds.

But autonomous doesn't mean unsupervised.

The Human-AI Partnership Model

When we tested AI-driven alert triage at Scanner, we ran an LLM against 348 known false positives and 1 true positive (a synthetic insider threat we crafted). The LLM classified each as Benign, Suspicious, or Malicious.

LLM alone: 71% accuracy. It correctly identified 247 as benign, but classified 51 as "Malicious" and 50 as "Suspicious." These were obvious false positives. The 51 false Malicious calls are brutal: each one interrupts an analyst for an urgent threat that doesn't exist. Cry wolf enough and analysts stop trusting the system. Worse, the LLM missed the one true positive entirely, calling our insider threat "benign."

With Socdown: 78% accuracy and zero false Malicious. We built a framework where the LLM follows hypothesis-driven methodology, surfaces evidence for review, and flags uncertain cases as Suspicious instead of guessing. It stopped crying wolf. And the insider threat? When an analyst prompted it to dig deeper, it generated 20+ investigative queries and correctly identified the threat.

The takeaway: LLMs can make decisions and accelerate investigations massively. But they shouldn't make sensitive changes autonomously. Give them freedom to investigate and make clear-cut calls, but route sensitive actions through human review.

Questions, not instructions. One pattern we've seen work well: have agents end their analysis with questions for the analyst rather than recommended next steps. "Disable this account" could cause damage if the agent is wrong. "Is this user authorized to access this system?" prompts investigation without prescribing action. Questions acknowledge uncertainty and keep the analyst in the driver's seat. Teams we've talked to found that

when agents give prescriptive instructions, analysts sometimes follow them without thinking—but questions force them to engage with the evidence and form their own conclusions.

Safe Agent Actions: Staging for Human Review

A critical principle when deploying autonomous agents: **use read-only tools for investigation, and staging tools for response.** Agents gather context and make recommendations, but consequential actions require human approval.

Think of it as a spectrum of trust with three categories:

Investigation Tools (Read-Only)

These actions are completely safe because they don't change anything. If the agent makes a mistake in its analysis, you've lost some time but caused no damage. Give agents full access to read-only investigation tools:

- **Security data lake queries** (Scanner, Splunk, Elastic): Search logs, analyze user behavior patterns, correlate activity across data sources
- **Threat intelligence** (VirusTotal, AbuseIPDB, threat feeds): Check IP/domain/file reputation, enrich alerts with external context
- **Cloud and endpoint context** (CrowdStrike, Wiz, AWS): Get device details, understand cloud resource relationships, verify configurations

Staging Tools (Human Review)

These create artifacts that humans review before anything irreversible happens. Agents take meaningful action while keeping humans in the decision loop:

- **Ticketing systems** (Jira, ServiceNow): Add comments with findings and recommendations, but humans decide whether to escalate or close
- **Communication** (Slack, Teams): Post triage summaries to channels, but humans decide the response
- **Code changes** (GitHub, GitLab): Open pull requests to update detection rules or blocklists, but humans review before merging
- **Drafts and queues**: Draft incident reports, queue credential rotations—but humans approve before execution

Response Tools (Avoid Initially)

Direct response actions are dangerous to automate because a misclassified alert could disable a legitimate user, block valid traffic, or disrupt production:

- Credential revocation
- Firewall modifications
- Account disabling
- Production config changes

Don't give agents these capabilities initially. Even as you build trust, consider whether they're ever appropriate for autonomous execution.

This graduated approach serves several purposes. It prevents irreversible damage from false positives. It creates an audit trail where every agent recommendation is documented for review. It keeps humans in the loop for high-stakes decisions. And it lets you build trust incrementally: start with read-only, graduate to staging, and only expand from there if the agent proves reliable.

Building Your First Triage Agent

Let's build a practical agent that pulls detection alerts from your security data lake, enriches them with context from multiple tools, and triages each one automatically.

Prerequisites

What you need

- A queryable security data lake with detection alerts (this example uses Scanner, but the pattern applies to any MCP-enabled data source)
- At least 30 days of user activity logs for baseline analysis
- Detection rules already generating alerts

How the Agent Connects to Your Tools

This agent uses the Model Context Protocol (MCP) to connect to your security tools. MCP is an open standard that lets AI agents interact with external systems through a consistent interface. Instead of writing custom API integrations, you configure MCP servers that expose your tools' capabilities to the agent.

The agent in this post connects to five tools:

Tool	Purpose	Required?
Scanner	Query logs, detection alerts, user behavior	Yes
VirusTotal	IP/domain/file reputation checks	No
CrowdStrike Falcon	Endpoint details, host health	No
Wiz	Cloud security issues, threats	No
Slack	Post triage summaries for review	No

You only need Scanner to get started. The other tools add context—uncomment them as you integrate each one.

Getting Your API Credentials

Anthropic API Key (required)
Sign up at console.anthropic.com and create an API key.

Scanner (required)
In your Scanner dashboard, go to Settings → API Keys and create a key with read access. Your MCP URL follows the pattern https://mcp.{your-env}.scanner.dev/v1/mcp.

VirusTotal (optional)
Create a free account at virustotal.com and find your API key in your profile settings. Free tier allows 4 requests/minute—if you're processing many alerts with external IPs, you'll hit this limit quickly. Consider a paid plan for production use, or have the agent batch and prioritize which IPs to check.

CrowdStrike Falcon (optional)
In the Falcon console, go to API Clients and Keys. Create an OAuth2 client with read permissions for Hosts and Detections. You'll get a client ID and secret. Your base URL depends on your cloud region (e.g., https://api.crowdstrike.com for US-1).

Wiz (optional)
In the Wiz console, go to Settings → Integrations → Wiz MCP. Create a new integration to get your client ID and secret. Wiz MCP runs as a container—see Wiz MCP docs for setup.

Slack (optional)
Create a Slack app at api.slack.com/apps with the chat:write and channels:read scopes. Install it to your workspace and copy the Bot User OAuth Token.

Project Setup

Dependencies (requirements.txt):

```
claude-agent-sdk
python-dotenv
rich
```

Environment variables (.env):

```
ANTHROPIC_API_KEY=your-anthropic-api-key
# Scanner (required)
SCANNER_MCP_URL=https://mcp.your-env-here.scanner.dev/v1/mcp
SCANNER_MCP_API_KEY=your-scanner-api-key

# VirusTotal (optional - for threat intel enrichment)
VIRUSTOTAL_API_KEY=your-virustotal-api-key

# CrowdStrike Falcon (optional - for endpoint context)
FALCON_CLIENT_ID=your-falcon-client-id
FALCON_CLIENT_SECRET=your-falcon-client-secret
FALCON_BASE_URL=https://api.crowdstrike.com

# Wiz (optional - for cloud security context, requires running Wiz MCP container)
WIZ_MCP_URL=http://localhost:8000/mcp

# Slack (optional - for posting triage summaries)
SLACK_BOT_TOKEN=xoxb-your-slack-bot-token
SLACK_TEAM_ID=your-slack-team-id
```

Install dependencies:

```
pip install -r requirements.txt
```

Install optional MCP servers (for tools you want to use):

```
# VirusTotal
npm install -g mcp-virustotal
```

```
# CrowdStrike Falcon
pip install falcon-mcp

# Slack
npm install -g @modelcontextprotocol/server-slack
```

Two Techniques That Make Agents More Reliable

Before we look at the code, it's worth understanding two techniques in the prompt that significantly improve agent accuracy.

Hypothesis-Driven Investigation. Instead of jumping to conclusions, the agent generates multiple hypotheses ranked by probability, then gathers evidence to test each one. This prevents tunnel vision. For any alert, the agent explicitly considers: What's the most likely benign explanation? What's the most likely malicious explanation? Are there other plausible scenarios? It then queries for specific evidence that would confirm or refute each hypothesis, evaluates what it finds, and classifies based on which hypothesis the evidence best supports. This mirrors how expert analysts think: consider alternatives, gather targeted evidence, update beliefs based on findings.

Self-Critique Loop. After reaching an initial conclusion, the agent critiques its own analysis, twice. It asks: What evidence might I have missed? Are there alternative explanations I didn't consider? Is my confidence level justified? What would change my classification? Running two critique passes matters because the first pass catches obvious gaps, while the second forces deeper reflection and sometimes reveals subtler issues. In our testing, agents with self-critique caught errors that uncritiqued agents missed, particularly false negatives where the agent initially dismissed real threats as benign.

The Triage Agent

This agent queries Scanner for new detection alerts, enriches them with context from multiple security tools, and triages each one. The prompt implements both techniques described above:

```
#!/usr/bin/env python3
import asyncio
import os
from datetime import datetime
from dotenv import load_dotenv
from rich import print as rprint
from claude_agent_sdk import query, ClaudeAgentOptions

async def run_triage_cycle():
	"""
	Pull recent detection alerts and triage each one using multiple security tools for
context.
	"""
	load_dotenv()

	options = ClaudeAgentOptions(
		model="claude-opus-4-5-20251101",
		allowed_tools=[
			# Scanner - security data lake (required)
			"mcp__scanner__get_scanner_context",
			"mcp__scanner__execute_query",
			"mcp__scanner__fetch_cached_results",
			# VirusTotal - threat intelligence (optional)
			"mcp__virustotal__get_ip_report",
			"mcp__virustotal__get_domain_report",
			"mcp__virustotal__get_file_report",
			"mcp__virustotal__get_url_report",
			# CrowdStrike Falcon - endpoint context (optional)
			"mcp__falcon__falcon_search_hosts",
			"mcp__falcon__falcon_get_host_details",
			"mcp__falcon__falcon_search_detections",
			"mcp__falcon__falcon_get_detection_details",
			# Wiz - cloud security context (optional)
			"mcp__wiz__wiz_get_issues",
			"mcp__wiz__wiz_get_issue_data_by_id",
			"mcp__wiz__wiz_get_projects",
			"mcp__wiz__wiz_get_threats",
			"mcp__wiz__wiz_search",
			# Slack - post triage summaries (optional)
			"mcp__slack__slack_list_channels",
			"mcp__slack__slack_post_message",
		],
		mcp_servers={
			# Required: Scanner for logs and detection alerts
			"scanner": {
			"type": "http",
			"url": os.environ.get("SCANNER_MCP_URL"),
			"headers": {
				"Authorization": f"Bearer {os.environ.get('SCANNER_MCP_API_KEY')}"
			}
			},
			# Optional: VirusTotal for threat intel
			# "virustotal": {
			#     "type": "stdio",
			#     "command": "mcp-virustotal",
			#     "env": {"VIRUSTOTAL_API_KEY": os.environ.get("VIRUSTOTAL_API_KEY")}
			# },
```

```
                    # Optional: CrowdStrike Falcon for endpoint context
                    # "falcon": {
                    #     "type": "stdio",
                    #     "command": "uvx",
                    #     "args": ["falcon-mcp"],
                    #     "env": {
                    #      "FALCON_CLIENT_ID": os.environ.get("FALCON_CLIENT_ID"),
                    #      "FALCON_CLIENT_SECRET": os.environ.get("FALCON_CLIENT_SECRET"),
                    #      "FALCON_BASE_URL": os.environ.get("FALCON_BASE_URL"),
                    #     }
                    # },
                    # Optional: Wiz for cloud security (requires running container)
                    # See https://docs.wiz.io/docs/set-up-wiz-mcp-server
                    # "wiz": {
                    #     "type": "http",
                    #     "url": os.environ.get("WIZ_MCP_URL"),
                    # },
                    # Optional: Slack for posting triage summaries
                    # "slack": {
                    #      "type": "stdio",
                    #      "command": "npx",
                    #      "args": ["-y", "@modelcontextprotocol/server-slack"],
                    #       "env": {
                    #             "SLACK_BOT_TOKEN": os.environ.get("SLACK_BOT_TOKEN"),
                    #                    "SLACK_TEAM_ID": os.environ.get("SLACK_TEAM_ID"),
                    #     }
                    # },
            }
        )       prompt = """
        You are a security alert triage agent. Your job is to:
        1. **Fetch recent alerts**: Query Scanner for recent detection alerts.
        Process them in severity order (Critical first, then High, Medium, Low).

        2. **For each alert**, use hypothesis-driven investigation:
                a. **Generate hypotheses**:
                        Based on the alert, generate 2-3 hypotheses ranked by probability:
                        - What's the most likely benign explanation?
                        - What's the most likely malicious explanation?
                        - Are there other plausible scenarios?
                b. **Gather targeted evidence** using all available tools:
                         - Query Scanner for related activity from the same user/IP
                        - Query Scanner for user's baseline behavior patterns
                        - Check VirusTotal for IP/domain/URL reputation (if available)
                        - Query CrowdStrike Falcon for endpoint context (if available)
                        - Query Wiz for cloud security issues or threats (if available)
                c. **Test hypotheses**: For each hypothesis, evaluate the evidence:
                        - Does the evidence support or contradict this explanation?
                        - Update your confidence in each hypothesis based on findings
                d. **Classify** based on which hypothesis the evidence best supports:
                        - BENIGN: Evidence strongly supports legitimate activity
                        - SUSPICIOUS: Evidence is mixed or insufficient, needs human review
                        - MALICIOUS: Evidence strongly supports malicious activity
        3. **Self-critique** (run this step twice):
        After your initial classification, critique your own analysis:
                - What evidence might you have missed?
                - Are there alternative explanations you didn't consider?
                - Is your confidence level justified by the evidence?
                - What would change your classification?

        Then revise your assessment if the critique reveals weaknesses.

        4. **Output a report for each alert** (Slack-friendly format):
        Always print the report. Use a format that works in Slack:
                - TL;DR: One-line summary with emoji, classification, and confidence  (BENIGN,
SUSPICIOUS,  MALICIOUS)
```

```
        - Key metadata on one line (user, severity, alert ID)
                - Key evidence as bullet points (not tables)
                - Questions for the analyst to investigate (not prescriptive next steps)

        Use *text* for bold, bullet points, and emojis for visual scanning.
        Keep it concise—analysts should grasp the situation in seconds.

        5. **Post to Slack** (if available):
        If Slack is connected, also post the report there.
        Important:
                - If no new alerts exist, report that and exit
                - Be conservative: when uncertain, classify as SUSPICIOUS rather than BENIGN
                - Note when you lack sufficient baseline data to make a confident assessment
                - The self-critique step is mandatory - always run it

        Start by querying for recent detection alerts.
        """

        rprint(f"[{datetime.now().isoformat()}] Starting triage cycle...")

        async for message in query(prompt=prompt, options=options):
                rprint(message)

        rprint(f"[{datetime.now().isoformat()}] Triage cycle complete.")

if __name__ == "__main__":
        asyncio.run(run_triage_cycle())
```

Each tool serves a distinct purpose in the investigation:

- **Scanner**: Query logs, analyze user behavior patterns, correlate activity across data sources
- **VirusTotal**: Check IP/domain/file/URL reputation against threat intelligence
- **CrowdStrike Falcon**: Search hosts and detections, get endpoint details, verify device health
- **Wiz**: Query security issues, search threats by CVE, get cloud resource and project context
- **Slack**: Post triage summaries for human review (staging tool)

This pattern mirrors what security teams are doing in production. For example, one team we spoke with runs agents that query Scanner, Wiz, and CrowdStrike Falcon together for every alert—automatically gathering context from all three before presenting findings to analysts.

What This Agent Does

The agent queries Scanner for detection alerts, enriches them with context from your other security tools, then works through each one using the hypothesis-driven approach. It generates hypotheses, gathers targeted evidence, tests each hypothesis against what it finds, runs two self-critique passes to catch blind spots, and outputs a detailed triage summary with its classification and reasoning.

Here's real output from an agent run, formatted for Slack:

```
SUSPICIOUS (Medium) - Team member assumed privileged access role

*User:* jchen | *Severity:* High | *Alert ID:* 7f3b2e1

*What happened:*
User assumed privileged role and executed AWS-RunShellScript on database bastion host.

*Why it's probably fine:*
• Known user with 973 events in the last 24 hours
• Source IP is known office IP (970K+ events, 9 users)
• User has assumed similar roles 6 times in past 2 weeks

*Why I'm still flagging it:*
• Shell script on database bastion - sensitive infrastructure
• Command parameters are redacted - can't verify what ran
• No way to check if this was authorized (no ticket system access)

*Questions for analyst:*
• Was this access authorized? Is there a change ticket?
• What commands were executed on the bastion host?
• Does this user normally access prod DB infrastructure?
```

The format is designed for quick scanning: emoji and classification up top, key context in the middle, questions at the bottom. An analyst can grasp the situation in seconds and knows exactly what to investigate next.

Running the Agent

```
python triage_agent.py
```

Tuning the Classification Thresholds

The prompt says "when uncertain, classify as SUSPICIOUS rather than BENIGN." This is a starting point. You'll iterate based on what you observe. If analysts are overwhelmed with too many SUSPICIOUS classifications, tighten the criteria and allow more BENIGN calls. If you're missing real threats, add more specific threat indicators for the agent to check. If specific alert types are misbehaving, add type-specific guidance to the prompt. Expect to revise this prompt multiple times as you learn how the agent behaves on your data.

Cost and Model Selection

This example uses claude-opus-4-5-20251101 for maximum reasoning capability, but that adds up at scale. Consider your options:

- **Claude Opus**: Best accuracy for complex, ambiguous alerts. Use for high-severity alerts or when you need deep reasoning.
- **Claude Sonnet**: Good balance of speed and accuracy. Suitable for most routine triage at lower cost.
- **Claude Haiku**: Fastest and cheapest. Use for high-volume, straightforward alerts where speed matters more than nuance.

A common pattern: route Critical/High severity alerts to Opus, and Medium/Low to Sonnet or Haiku. You can also start with Opus to establish a baseline, then test cheaper models to see if accuracy holds.

Key Takeaways

Give agents autonomy for routine work, but stage sensitive actions for human review. Let agents query data and triage clear-cut cases autonomously. For ambiguous alerts or consequential actions (ticket creation, config changes), have humans approve. In our testing, unguided AI missed critical threats that human-guided AI caught.

Use hypothesis-driven investigation. Having agents generate and test multiple hypotheses prevents tunnel vision and produces more reliable classifications than jumping straight to conclusions.

Add self-critique loops. Two passes of self-critique catch blind spots and overconfidence. The agent should ask itself what evidence it might have missed and what would change its classification.