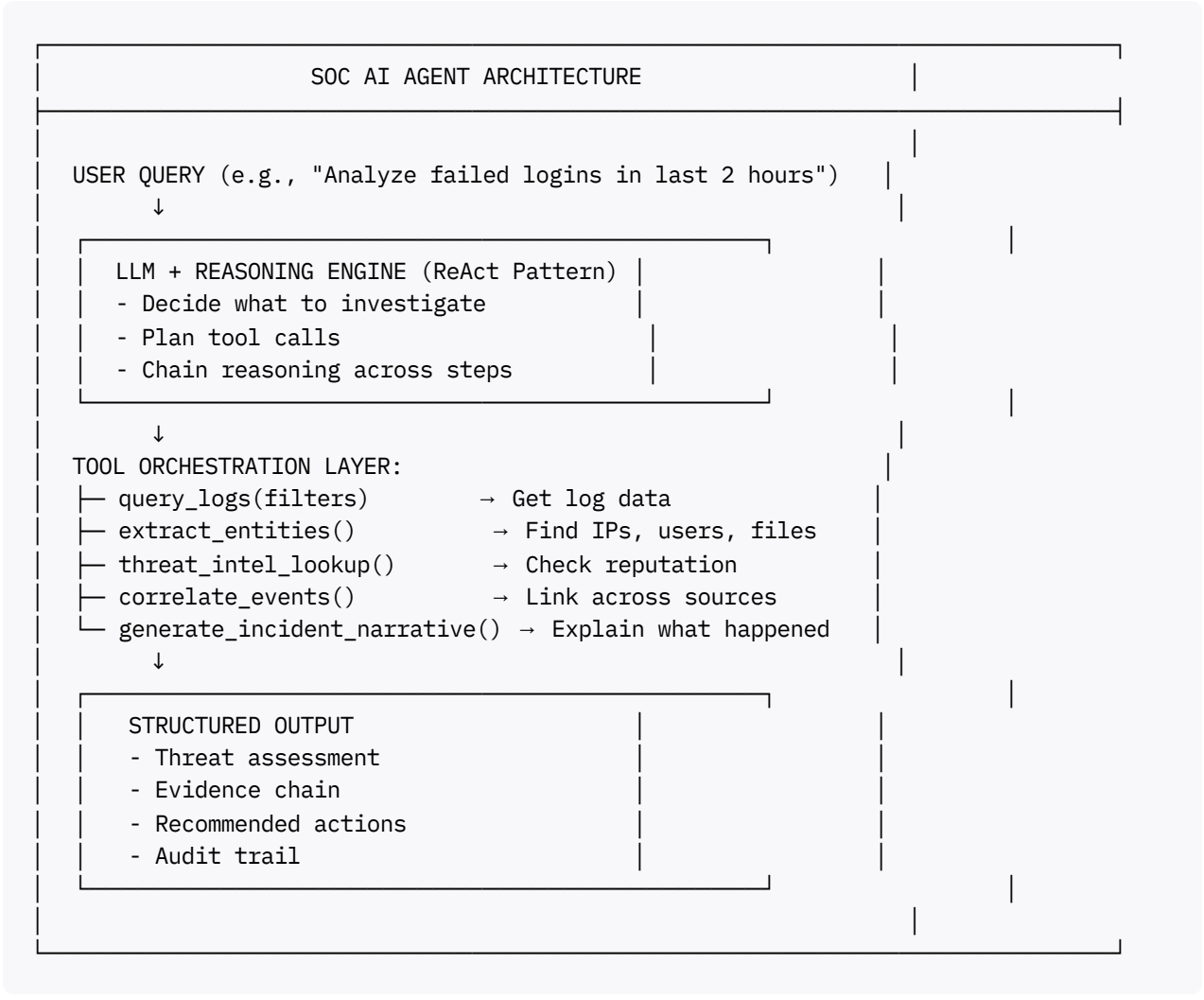
perplexity

Building an AI Agent for SOC Log Analysis: Complete Step-by-Step Guide

I'll walk you through building a **production-ready AI agent** for SOC log analysis using modern frameworks. This is what winning competition teams are building.

Architecture Overview

Your AI agent needs these components (in order):



Step 1: Set Up Your Environment

Install dependencies:

```
pip install langchain langchain-community langchain-openai \
            langgraph pydantic pandas numpy \
            python-dotenv requests
```

Optional (for local models):

```
# For running open-source models locally
pip install ollama
# Then: ollama pull llama2 (or mistral, neural-chat, etc.)
```

Step 2: Define Your "Tools"

Tools are functions the agent can call autonomously. These are your critical building blocks:

```
from langchain.tools import tool
from typing import Optional, List
import pandas as pd
import json

# ============================================
# TOOL 1: Query Logs
# ============================================
@tool
def query_logs(
    filters: dict,
    time_range_hours: int = 2,
    limit: int = 1000
) -> str:
    """
    Query security logs from your SIEM/database.

    Args:
        filters: e.g., {"event_type": "auth_failure", "source_ip": "192.168.1.100"}
        time_range_hours: Look back N hours
        limit: Max results to return

    Returns:
        JSON string of log records
    """
    # TODO: Replace with your actual log storage query
    # Options: Elasticsearch, Splunk API, ClickHouse, PostgreSQL, etc.

    # EXAMPLE (mock):
    mock_logs = [
        {
            "timestamp": "2025-01-15T10:23:45Z",
```

```
            "event_type": "auth_failure",
            "user": "admin",
            "source_ip": "192.168.1.100",
            "attempt_count": 50,
            "service": "ssh"
        },
        {
            "timestamp": "2025-01-15T10:25:12Z",
            "event_type": "auth_success",
            "user": "admin",
            "source_ip": "192.168.1.100",
            "service": "ssh"
        }
    ]

    # Real implementation: Filter based on filters dict, time_range_hours, limit
    # For now, return mock
    return json.dumps(mock_logs, indent=2)


# ============================================
# TOOL 2: Extract & Enrich Entities
# ============================================
@tool
def extract_entities(logs_json: str) -> str:
    """
    Extract security entities from logs and enrich with threat intel.

    Args:
        logs_json: JSON string of logs

    Returns:
        JSON with extracted entities and threat scores
    """
    logs = json.loads(logs_json)
    entities = {
        "ips": [],
        "users": [],
        "files": [],
        "domains": []
    }

    for log in logs:
        if "source_ip" in log:
            ip = log["source_ip"]
            # MOCK: Check threat intel (replace with real API call)
            threat_score = check_threat_intel(ip)
            entities["ips"].append({
                "value": ip,
                "threat_score": threat_score,
                "is_malicious": threat_score > 0.7
            })

        if "user" in log:
            entities["users"].append(log["user"])
```

```
    return json.dumps(entities, indent=2)


# ============================================
# TOOL 3: Threat Intelligence Lookup
# ============================================
@tool
def threat_intel_lookup(entity: str, entity_type: str) -> str:
    """
    Look up an entity (IP, domain, file hash) in threat intel databases.

    Args:
        entity: The IP, domain, or hash
        entity_type: "ip", "domain", or "file_hash"

    Returns:
        JSON with threat info and reputation
    """
    # EXAMPLE: You'd call VirusTotal, AlienVault OTX, AbuseIPDB, etc.
    # For now, mock:

    threat_data = {
        "entity": entity,
        "type": entity_type,
        "reputation": "unknown",
        "threat_score": 0.0,
        "known_malicious": False,
        "last_seen": None,
        "sources": []
    }

    # REAL IMPLEMENTATION (example with VirusTotal):
    # import requests
    # response = requests.get(
    #     f"https://www.virustotal.com/api/v3/ip_addresses/{entity}",
    #     headers={"x-apikey": VIRUSTOTAL_API_KEY}
    # )
    # threat_data = response.json()

    return json.dumps(threat_data, indent=2)


# ============================================
# TOOL 4: Correlate Events
# ============================================
@tool
def correlate_events(logs_json: str) -> str:
    """
    Find correlations across multiple logs.
    Links auth events -> endpoint events -> network events.

    Args:
        logs_json: JSON string of logs

    Returns:
        Correlated incident narrative
```

```
    """
    logs = json.loads(logs_json)

    # Group by user/IP to find patterns
    patterns = {}
    for log in logs:
        key = f"{log.get('user')}_{log.get('source_ip')}"
        if key not in patterns:
            patterns[key] = []
        patterns[key].append(log)

    # Detect suspicious sequences
    correlations = {
        "sequences_found": [],
        "attack_hypotheses": []
    }

    for key, events in patterns.items():
        # EXAMPLE: Multiple failed logins followed by success = brute force
        failures = [e for e in events if e.get("event_type") == "auth_failure"]
        successes = [e for e in events if e.get("event_type") == "auth_success"]

        if len(failures) > 10 and len(successes) > 0:
            correlations["attack_hypotheses"].append({
                "pattern": "Brute Force Attack",
                "evidence": f"{len(failures)} failed attempts before success",
                "confidence": 0.85,
                "actors": [key]
            })

    return json.dumps(correlations, indent=2)


# ============================================
# TOOL 5: Generate Incident Narrative
# ============================================
@tool
def generate_incident_narrative(
    logs_json: str,
    entities_json: str,
    correlations_json: str
) -> str:
    """
    Generate a plain-English narrative of what happened.

    Args:
        logs_json: Original logs
        entities_json: Extracted entities
        correlations_json: Correlated patterns

    Returns:
        Human-readable incident summary
    """
    logs = json.loads(logs_json)
    entities = json.loads(entities_json)
    correlations = json.loads(correlations_json)
```

```
    narrative = f"""
    INCIDENT SUMMARY
    ================

    Time Range: {logs[^0]['timestamp'] if logs else 'N/A'} to {logs[-1]['timestamp'] if l

    ACTORS INVOLVED:
    - Users: {', '.join(set(e.get('user', 'unknown') for e in logs))}
    - IPs: {', '.join(entities.get('ips', []))}

    ATTACK HYPOTHESIS:
    """

    for hypothesis in correlations.get("attack_hypotheses", []):
        narrative += f"\n    - {hypothesis['pattern']} (Confidence: {hypothesis['confider
        narrative += f"\n      Evidence: {hypothesis['evidence']}"

    narrative += "\n    \n    RECOMMENDED ACTIONS:\n"
    narrative += "    1. Investigate source IP for credential compromise\n"
    narrative += "    2. Reset password for affected user\n"
    narrative += "    3. Check for lateral movement in endpoint logs\n"

    return narrative


# ============================================
# HELPER FUNCTION
# ============================================
def check_threat_intel(ip: str) -> float:
    """Mock threat intel lookup. Return threat score 0.0-1.0"""
    # In reality, call VirusTotal, AbuseIPDB, etc.
    # For now, hardcode some known malicious IPs
    if ip in ["1.2.3.4", "5.6.7.8"]:
        return 0.95
    return 0.1
```

Step 3: Create the Agent Using LangGraph (ReAct Pattern)

LangGraph is the modern way to build agentic systems. It gives you control over the agent's reasoning loop:

```
from langchain_openai import ChatOpenAI
from langchain.agents import tool
from langgraph.graph import END, StateGraph
from typing import TypedDict, Annotated
import operator

# ============================================
# Define Agent State
# ============================================
class AgentState(TypedDict):
    messages: Annotated[list, operator.add]
```

```
    logs: str
    entities: str
    correlations: str
    narrative: str
    next_action: str


# ============================================
# Initialize LLM
# ============================================
# Option 1: OpenAI (requires API key)
llm = ChatOpenAI(model="gpt-4o", temperature=0.3)

# Option 2: Local Llama (free, runs on your machine)
# from langchain_community.llms import Ollama
# llm = Ollama(model="mistral")


# ============================================
# Define Agent Reasoning Node
# ============================================
def reasoning_node(state: AgentState):
    """
    The LLM decides what to do next based on conversation history.
    This is where "thinking" happens.
    """
    system_prompt = """
    You are an expert SOC analyst powered by AI. Your job is to analyze security logs and

    You have access to these tools:
    - query_logs: Get logs from the SIEM
    - extract_entities: Find IPs, users, files in logs
    - threat_intel_lookup: Check if an IP/domain is malicious
    - correlate_events: Link events across sources to find patterns
    - generate_incident_narrative: Create a human-readable report

    When analyzing logs, follow this process:
    1. Query logs based on what the user asked
    2. Extract entities and enrich with threat intelligence
    3. Correlate events to find attack patterns
    4. Generate a narrative explaining what happened
    5. Provide confidence scores and recommended actions

    Always cite evidence from the raw logs when making claims.
    """

    # Build message history
    messages = state.get("messages", [])

    # Call LLM to decide next action
    response = llm.invoke([
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": messages[-1] if messages else "Analyze security logs
    ])

    return {
```

```
        "messages": messages + [response],
        "next_action": "query_logs"  # Simplified; in reality, parse LLM output
    }


# ============================================
# Define Tool-Calling Nodes
# ============================================
def query_logs_node(state: AgentState):
    """Execute log query tool"""
    result = query_logs({"event_type": "auth_failure"}, time_range_hours=2)
    return {
        "logs": result,
        "next_action": "extract_entities"
    }


def extract_entities_node(state: AgentState):
    """Execute entity extraction"""
    result = extract_entities(state["logs"])
    return {
        "entities": result,
        "next_action": "correlate_events"
    }


def correlate_events_node(state: AgentState):
    """Execute event correlation"""
    result = correlate_events(state["logs"])
    return {
        "correlations": result,
        "next_action": "generate_narrative"
    }


def generate_narrative_node(state: AgentState):
    """Generate final incident report"""
    result = generate_incident_narrative(
        state["logs"],
        state["entities"],
        state["correlations"]
    )
    return {
        "narrative": result,
        "next_action": END
    }


# ============================================
# Build the Agent Graph
# ============================================
workflow = StateGraph(AgentState)

# Add nodes
workflow.add_node("reasoning", reasoning_node)
workflow.add_node("query_logs", query_logs_node)
```

```
workflow.add_node("extract_entities", extract_entities_node)
workflow.add_node("correlate_events", correlate_events_node)
workflow.add_node("generate_narrative", generate_narrative_node)

# Add edges (define flow)
workflow.add_edge("reasoning", "query_logs")
workflow.add_edge("query_logs", "extract_entities")
workflow.add_edge("extract_entities", "correlate_events")
workflow.add_edge("correlate_events", "generate_narrative")
workflow.add_edge("generate_narrative", END)

# Set entry point
workflow.set_entry_point("reasoning")

# Compile
agent = workflow.compile()
```

Step 4: Run the Agent

```
# ============================================
# Execute the Agent
# ============================================
def run_soc_analysis(user_query: str):
    """Run the agent with a user query"""

    print(f"\n USER QUERY: {user_query}\n")

    # Initialize state
    initial_state = {
        "messages": [user_query],
        "logs": "",
        "entities": "",
        "correlations": "",
        "narrative": "",
        "next_action": "reasoning"
    }

    # Execute agent
    result = agent.invoke(initial_state)

    # Display results
    print("\n" + "="*60)
    print("INCIDENT ANALYSIS REPORT")
    print("="*60)
    print(result["narrative"])
    print("\n" + "="*60)
    print("RAW EVIDENCE")
    print("="*60)
    print("\nLOGS:")
    print(result["logs"])
    print("\nENTITIES:")
    print(result["entities"])
    print("\nCORRELATIONS:")
    print(result["correlations"])
```

```
# Example usage
if __name__ == "__main__":
    run_soc_analysis("Analyze failed login attempts in the last 2 hours")
```

Step 5: Add Tool-Calling (Advanced)

For a more autonomous agent, use **tool calling** to let the LLM decide which tools to use:

```
from langchain.agents import tool
from langchain.agents.format_scratchpad import format_to_openai_function_calls
from langchain.agents.output_parsers import OpenAIFunctionsAgentOutputParser

# Define tools for the agent
tools = [
    query_logs,
    extract_entities,
    threat_intel_lookup,
    correlate_events,
    generate_incident_narrative
]

# Create tool bindings
llm_with_tools = llm.bind_functions(tools)

# Agent loop
def agent_loop(user_query: str):
    messages = [{"role": "user", "content": user_query}]

    while True:
        # LLM decides which tool to call
        response = llm_with_tools.invoke(messages)

        # If it's a tool call, execute it
        if hasattr(response, "tool_calls") and response.tool_calls:
            for tool_call in response.tool_calls:
                tool_name = tool_call["function"]["name"]
                tool_args = tool_call["function"]["arguments"]

                # Execute the tool
                tool_result = globals()[tool_name](**tool_args)

                # Add to message history
                messages.append({"role": "assistant", "content": str(response)})
                messages.append({
                    "role": "user",
                    "content": f"Tool {tool_name} returned: {tool_result}"
                })
        else:
            # LLM provided final answer
            print("FINAL REPORT:")
            print(response.content)
            break
```

```
# Run
agent_loop("Investigate failed logins from 1.2.3.4")
```

Step 6: Integration Points (Where to Connect Real Data)

Replace these mock implementations with your actual SOC infrastructure:

Component	Replace With
`query_logs()`	Elasticsearch, Splunk API, or SIEM SQL interface
`threat_intel_lookup()`	VirusTotal, AbuseIPDB, AlienVault OTX API
`correlate_events()`	Your log correlation engine or SIEM
`check_threat_intel()`	Your internal threat database or API

Example: Real Elasticsearch Integration

```
from elasticsearch import Elasticsearch

es = Elasticsearch(["http://localhost:9200"])

@tool
def query_logs_real(filters: dict, time_range_hours: int = 2):
    """Query real logs from Elasticsearch"""
    query = {
        "query": {
            "bool": {
                "must": [
                    {"range": {"timestamp": {"gte": f"now-{time_range_hours}h"}}},
                    *[{"match": {k: v}} for k, v in filters.items()]
                ]
            }
        },
        "size": 1000
    }

    results = es.search(index="security-logs-*", body=query)
    return json.dumps(results["hits"]["hits"], indent=2)
```

Step 7: Add Structured Output & Audit Trail

For judges to evaluate your work, output structured JSON:

```
from pydantic import BaseModel
from typing import List

class ThreatEvidence(BaseModel):
    log_line_number: int
```

```
    log_content: str
    why_suspicious: str

class IncidentReport(BaseModel):
    title: str
    severity: str  # "critical", "high", "medium", "low"
    confidence: float  # 0.0-1.0
    summary: str
    attack_hypothesis: str
    evidence: List[ThreatEvidence]
    recommended_actions: List[str]
    false_positive_risk: str  # Why this might be benign


# Output example
report = IncidentReport(
    title="Brute Force Attack on Admin Account",
    severity="critical",
    confidence=0.92,
    summary="50 failed SSH login attempts followed by success from 1.2.3.4",
    attack_hypothesis="Attacker brute-forced the admin password and gained access",
    evidence=[
        ThreatEvidence(
            log_line_number=1,
            log_content="Failed password for user admin from 192.168.1.100 port 22 ssh",
            why_suspicious="Multiple failed attempts in short time"
        ),
        ThreatEvidence(
            log_line_number=51,
            log_content="Accepted password for user admin from 192.168.1.100 port 22 ssh'
            why_suspicious="Success after 50 failures suggests successful brute force"
        )
    ],
    recommended_actions=[
        "Reset admin password immediately",
        "Check for lateral movement from this IP",
        "Review command history for admin user",
        "Block this IP at firewall"
    ],
    false_positive_risk="Low: Pattern clearly matches brute force attack. Only false if I
)

print(report.model_dump_json(indent=2))
```

Step 8: Performance Optimization

For competition judges, show you can handle real-world scale:

```
import time
from functools import lru_cache

# Cache threat intelligence lookups (same IP won't change in minutes)
@lru_cache(maxsize=10000)
def threat_intel_lookup_cached(entity: str, entity_type: str) -> str:
```

```
    """Cached threat intel to reduce API calls"""
    return threat_intel_lookup(entity, entity_type)

# Batch processing
def process_logs_batch(log_list: List[str]):
    """Process multiple log files in parallel"""
    from concurrent.futures import ThreadPoolExecutor

    start_time = time.time()

    with ThreadPoolExecutor(max_workers=4) as executor:
        results = list(executor.map(run_soc_analysis, log_list))

    elapsed = time.time() - start_time
    print(f"Processed {len(log_list)} incidents in {elapsed:.2f}s ({len(log_list)/elapsed

    return results
```

Step 9: Testing & Demo (What Judges See)

Create a simple CLI or web UI:

```
# streamlit_demo.py
import streamlit as st

st.title("SOC AI Agent")

user_query = st.text_input("Describe the incident you want to analyze:")

if st.button("Analyze"):
    with st.spinner("Analyzing logs..."):
        report = run_soc_analysis(user_query)

    st.json(report.model_dump())
    st.download_button(
        "Download Report",
        report.model_dump_json(indent=2),
        file_name="incident_report.json"
    )
```

Run with: `streamlit run streamlit_demo.py`

Key Takeaways for Winning

1. **Start with LangGraph** – It's the modern standard for agentic systems (vs older ReAct implementations)
2. **Tool orchestration matters** – Good agents chain tools logically (query → extract → correlate → narrate)
3. **Explainability is mandatory** – Every finding must cite log line numbers

4. **Handle false positives** – Acknowledge why benign activity might look suspicious
5. **Real data beats demos** – Connect to actual SIEM/logs, not mocks
6. **Structured output** – JSON schemas make judging easier
7. **Latency metrics** – Show you can process large log volumes fast
8. **Audit trails** – Log every decision the agent makes for transparency

This architecture is what leading security companies (RunReveal, Exabeam, Rapid7) are building today.[1] [2] [3]

⁂

1. https://blog.runreveal.com/how-we-built-model-agnostic-ai-agent-log-analysis/
2. https://www.gocodeo.com/post/agentic-ai-for-cybersecurity-autonomous-threat-detection-and-response
3. https://jisem-journal.com/index.php/journal/article/view/10967
4. https://www.ibm.com/think/tutorials/using-langchain-tools-to-build-an-ai-agent
5. https://devsec-blog.com/2024/12/building-ai-agents-to-solve-security-challenges/
6. https://www.linkedin.com/pulse/autonomous-ai-agents-cybersecurity-real-time-threat-detection-feaxc
7. https://realm.security/security-monitoring-for-ai-agents-and-mcp/
8. https://www.langchain.com/langchain
9. https://www.balbix.com/insights/understanding-agentic-ai-and-its-cybersecurity-applications/
10. https://www.exabeam.com/explainers/ai-cyber-security/the-ai-powered-soc-capabilities-benefits-and-best-practices/