AI Sentinel: Code Workflow Deep Dive

This document provides an exhaustive look at the internal logic and inter-file communication within the AI Sentinel platform.

Core Module Interaction

AI Sentinel follows a modular architecture where detection, orchestration, and action are decoupled.

1. The Entry Point: `app.py`

'app.py' serves as the Streamlit-based UI and the main controller.

- Workflow:
- Loads logs from 'data/sample_logs.json' or user upload.
- Initializes 'ShadowAIDetector' from 'src/detector.py'.
- Calls 'detector.batch_analyze(logs, use_agent=True)'.
- Results are stored in 'st.session_state' and passed to rendering functions:
- 'render_metrics()': Aggregate stats.
- 'render_behavioral_analytics()': Charts via 'detector.get_analytics()'.
- 'render_action_center()': Interactive manual review.
- 'SecurityAdvisor': Virtual chat assistant.

2. The Hub: `src/detector.py`

The 'ShadowAIDetector' class manages the high-level detection pipeline.

- 'analyze_request(log)':
  - Step A: Pre-screening: Calls 'self._pre_screening(log)'. This checks 'MALICIOUS_DOMAINS' and runs 'SENSITIVE_PATTERNS' regex scanning defined in 'src/policies.py'.
- Step B: Agentic Orchestration: If 'use_agent' is True, it instantiates 'SecurityAgent' and calls '.run(log)'.
- Step C: Results Merging: Combines static findings and agentic reasoning into a final structured dict.
  - 'get_analytics()': Uses Pandas to transform raw result lists into departmental risk heatmaps and exfiltration metrics.

3. The Brain: `src/agents.py`

This module contains the sophisticated LangGraph-based logic.

- 'SecurityAgent':
- Graph Design: 'analyzer' Node -> 'mitigator' Node.
  - 'analyzer': Passes the log + policy context to the LLM. It supports Deterministic Overridesif pre-screening found a confirmed threat, it forces the risk to 'CRITICAL' regardless of LLM output.
- 'mitigator': Evaluates the risk category. If 'CRITICAL' or 'HIGH_RISK', it calls external mitigation modules.
- 'SecurityAdvisor':

- A standalone conversational class. It takes the current 'results' list as context and answers user queries about specific threats or general policy.

4. The Action Layer: `src/webhooks.py` & `src/notifications.py`

These modules handle the side-effects of the security assessment.

- 'WebhookManager.trigger_mitigation()': Simulates REST API calls to block IP addresses on a firewall.
- 'broadcast_alert()': Formats security alerts and "sends" them to Slack and Microsoft Teams via mock functions.

5. The Policy Layer: `src/policies.py`

Centralized configuration for the entire system.

- Data: IBAN regexes, phone patterns, department risk levels, and known malicious URL lists.
- Prompting: Contains 'get_detection_prompt()', which generates the system instructions for the LLM, ensuring it acts as a professional SOC analyst.

Final Execution Flow Diagram

1. User Upload ('app.py')
2. Batch Process ('detector.py')
3. Regex/Static Check ('policies.py')
4. Agentic Reasoning ('agents.py' -> LLM)
5. Mitigation Trigger ('webhooks.py' / 'notifications.py')
6. Analytics Aggregation ('detector.py' -> Pandas)
7. UI Render ('app.py' UI Components)

Internal Technical Reference - Phase 3 Final Development