Automaton Auditor: Interim Project Report

Date: February 25, 2026

Project: Week 2 FDE Challenge – The Automaton Auditor

Author: Yakob Dereje

---

1. Executive Summary

The Automaton Auditor is a multi-agent system built on LangGraph designed to perform autonomous forensic audits of GitHub repositories. The system focuses on identifying the presence of specific technical implementations (LangGraph usage), documentation standards (PDF coverage), and development best practices (Atomic Git history). This report outlines the current architectural state, design justifications, and the roadmap for the judicial synthesis layer.

---

2. Architecture Decisions

A. Pydantic Models vs. Standard Dictionaries

We opted to define our AgentState and Evidence objects using Pydantic rather than raw Python dictionaries.

- Data Integrity: Pydantic enforces strict type validation at every node transition. This ensures that the "Fan-In" aggregation stage never receives malformed evidence that could crash the Judge node.
- Schema Enforcement: By using a structured Evidence class (containing title, summary, severity, and verdict), we ensure that regardless of which detective node produces the finding, the data remains consistent and comparable.

B. AST-Based Structural Analysis

Instead of relying on simple string matching or regex—which are prone to false positives—our RepoInvestigator utilizes Python's Abstract Syntax Tree (AST) module.

- Logic Verification: The auditor parses the source code into a tree structure to identify formal StateGraph assignments and node definitions.
- Reliability: This approach allows the system to distinguish between actual functional code and commented-out snippets or plain text mentions of "LangGraph," ensuring a high-fidelity audit.

C. Sandboxing Strategy

To maintain security and environmental isolation, we implemented a dedicated clone_repo node.

- Temporary Workspaces: Repositories are cloned into randomized temporary directories using tempfile. This prevents local file system pollution and ensures each audit starts from a "clean slate".
- Isolation: All subsequent forensic tools (Git extraction and AST parsing) operate exclusively within this restricted directory path, ensuring the host system remains untouched by external code.

---

3. Planned StateGraph Flow

The system utilizes a Diamond Architecture (Parallel Fan-Out/Fan-In), which allows for high-efficiency auditing by running independent forensic tasks simultaneously.

Visual Logic Diagram

Node Descriptions

1. START: Accepts the initial state containing the repo_url.
2. Clone Node: Performs the sandboxed cloning operation and updates the state with the repo_path.
3. Detective Fan-Out:
   - Repo Investigator: Analyzes Git history for atomic commits and performs AST parsing for LangGraph usage.
   - Doc Analyst: Uses Docling to scan the repository for required PDF documentation.
4. Fan-In (Aggregation): Both detectives append their findings to a shared evidences list using a reducer (operator.add), ensuring data from both branches is preserved.
5. Judge Node: Processes the aggregated evidence list to calculate a final weighted score (e.g., 70/100) and issues a final verdict.
6. END: Returns the final audited state.

---

4. Known Gaps and Concrete Roadmap

A. Current Gaps

- Static Limitation: The current system primarily performs "Existence Checks" (e.g., "Is LangGraph imported?" or "Are there PDFs?") rather than evaluating quality.
- Deterministic Judging: The current Judge node uses basic weighted math rather than nuanced reasoning.

B. Phase 2: The Judicial Layer & Synthesis Engine

- LLM-Powered Judicial Layer: We plan to integrate a "Reasoning Agent" powered by Gemini or OpenAI. This node will evaluate the *context* of the evidence—for example, reading commit messages to determine if they actually describe the changes made, rather than just counting the number of commits.
- Synthesis Engine: We will implement a final node that translates the raw Evidence list into a human-readable PDF summary report, providing students with constructive feedback on how to improve their score.

- Dynamic Simulation: A future "Simulation Node" will attempt to dry-run the student's graph to verify if it actually executes, moving beyond static analysis to dynamic verification.

---

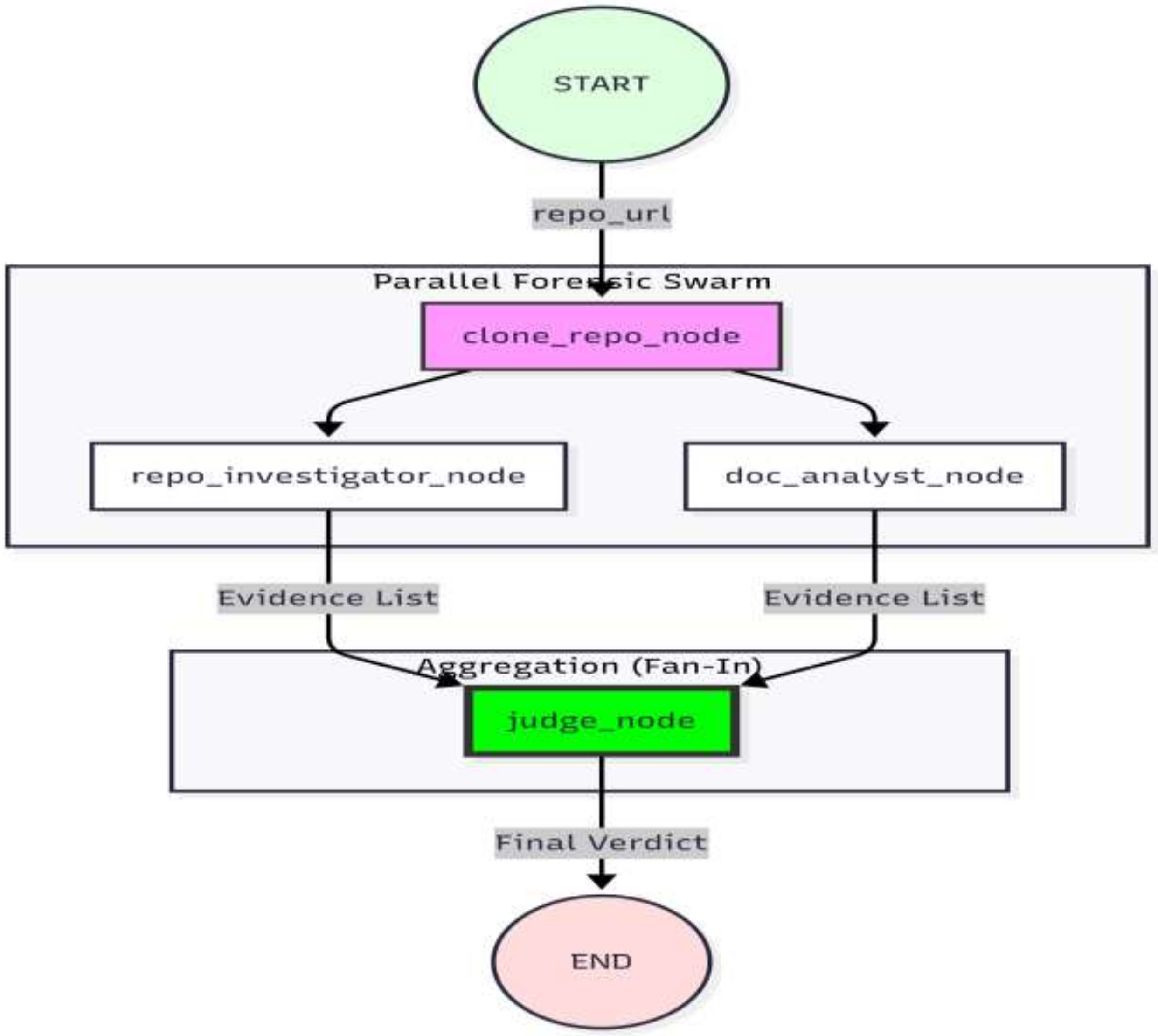
5. Technical Stack Summary

- Orchestration: LangGraph.
- Forensics: AST (Python), GitPython.
- Doc Processing: Docling.
- Environment Management: UV.

---

6. Diagrams showing the planned StateGraph flow (detective fan-out/fan-in)

```
graph TD

    START((START)) -->|repo_url| Clone[clone_repo_node]

    subgraph "Parallel Forensic Swarm (Fan-Out)"

    Clone --> Repo[repo_investigator_node]

    Clone --> Doc[doc_analyst_node]

    End

    subgraph "Aggregation (Fan-In)"

    Repo -->|Evidence List| Judge[judge_node]

    Doc -->|Evidence List| Judge

    end

    Judge -->|Final Verdict| END((END))

    style Clone fill:#f9f,stroke:#333,stroke-width:2px

    style Judge fill:#00ff00,stroke:#333,stroke-width:4px

    style START fill:#dfd

    style END fill:#fdd
```



Technical Flow Explanation:

- Fan-Out (Detective Parallelism): To minimize execution latency, the system triggers the RepoInvestigator and DocAnalyst simultaneously. This allows the system to perform heavy AST parsing and PDF scanning in parallel, reducing the bottleneck of serial processing.
- Fan-In (Evidence Synthesis): Both branches converge at the Judge node. This "Fan-In" pattern is critical because it ensures the Judge waits for all forensic data to be collected before calculating the final audit score. This prevents a "partial pass" scenario where a verdict is issued before all evidence is submitted.
- State Management: This flow is made possible by the Annotated[list, operator.add] reducer in the AgentState, which allows independent nodes to "add" their findings to a growing list without overwriting the data from the other branch.