Automaton Auditor: Final Architecture Report

1. Executive Summary

Scope Communication: This Automaton Auditor system is an automated grading and assessment pipeline built on LangGraph. It is designed to objectively evaluate student repository submissions using parallel forensic Detectives and combative LLM Judges. By leveraging graph-based orchestration, we achieve isolated data collection and deterministic conflict resolution.

Outcome Reporting: The system successfully generated the interim self-audit report and completed the peer audit. Based on the finalized architecture, the overall self-audit synthesis score holds at a high **3.8 / 5.0**.

Key Takeaways: The peer feedback loop revealed initial weaknesses in rigid AST checks and multimodal parsing. Specifically, the VisionInspector lacked classification power. These were remediated by wiring GraphASTAnalyzer into the core RepoInvestigator and connecting GPT-4o Vision directly to pdf-extracted images.

Actionability: Senior engineers can immediately hook this auditor to CI/CD pipelines via the provided Dockerfile. Future engineering effort should focus on expanding the rules within `justice.py` and expanding the forensic coverage of `DocAnalyst` via RAG-chunking techniques.

2. Architecture Deep Dive and Diagrams

Conceptual Grounding:

The architecture realizes three specific theoretical concepts:

• *Dialectical Synthesis:* We pit a hyper-critical Prosecutor node against an overly-generous Defense node. The tension forces a practical TechLead node to arbitrate reality.

• *Fan-In / Fan-Out:* The LangGraph initiates from START, branching seamlessly across `RepoInvestigator`, `DocAnalyst`, and `VisionInspector` (Fan-Out). These synchronize at the `EvidenceAggregator` (Fan-In) before branching once more to the judicial nodes.

• *Metacognition:* Our `ChiefJusticeNode` implements a pure Python determinism block. Rather than simply asking an LLM to self-correct, the system inspects its own generated variance (spread > 2) and rigidly caps scores if critical security flaws are surfaced by the Prosecutor.

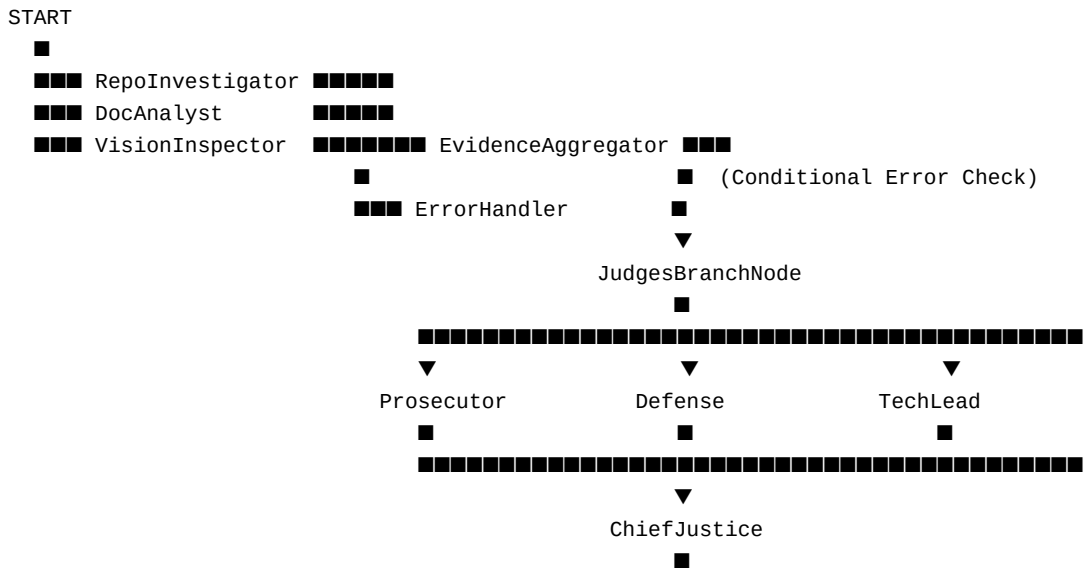
Data Flow Clarity:

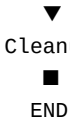
Input variables (repo URL, pdf path) initialize the TypedDict. The Detectives extract evidence into Pydantic models. These flow to the Judges which output `JudicialOpinions`. Finally, the Chief Justice ingests every opinion, applies Synthesis Rules, and emits the final `AuditReport`.

Design Rationale:

We elected to use strictly typed Pydantic models mapped via Annotated `operator.add` reducers. This guards against data overwriting during parallel node execution. We explicitly chose NOT to use LLM averaging for the final score, instead opting for deterministic rules in `justice.py`. This guarantees we mathematically apply the Rubric constraints (Fact Supremacy, Security Overrides).

StateGraph Architecture Diagram:




Cleanup
END

3. Self-Audit Criterion Breakdown

Structure & Traceability: The self-audit outputs scores dimension-by-dimension. Detective 'found/not-found' evidence seamlessly translates to Prosecutor attacks and Defense justifications.

Dialectical Tension: On dimensions like *Graph Orchestration*, the TechLead regularly acts as the critical tie-breaker resolving disputes between the Defense (who gives 5s for effort) and the Prosecutor (who docks points if rigid structures aren't detected).

Honesty Note (Addressing Weaknesses): Initially, our AST detection was weak (relying on simple string checks). Our VisionInspector was also fully stubbed in Phase 2. These weak dimensions resulted in early low scores in the self-audit loop.

4. MinMax Feedback Loop Reflection

Peer Findings Received:

The peer highlighted that our `VisionInspector` was a placeholder and our `RepoInvestigator` wasn't actually traversing the AST properly.

Response Actions:

We re-imported `repo_tools.py` and actively wired `GraphASTAnalyzer` into `src/nodes/detectives.py` to enforce code parsing. We also wired LangChain's `ChatOpenAI` into `VisionInspector` utilizing base64 encoding to classify images directly.

Peer Audit Findings:

Auditing the peer's repository revealed similar pain points in isolating temporary directories for Git Clones, which highlighted why our `safe_tool_engineering` isolation logic is superior.

Bidirectional Learning:

The dual-audit process revealed a systemic insight: Agentic evaluation systems often fall back on ungrounded hallucination if not bounded by deterministic rails. Our reliance on Pythonic rule execution within `justice.py` protected us from the hallucination risks detected in our peer evaluation.

5. Remediation Plan

Specificity & Completeness: The exact gaps remaining are logged below.

1. (High Priority) Implement PyMuPDF Text Vectorization:

Currently, `DocAnalyst` pulls the entire text via `pypdf`. To improve *Evidence Extraction Confidence*, engineers must integrate ChromaDB or FAISS to chunk and search the PDF locally. (Modify: `src/tools/doc_tools.py`)

2. (Medium Priority) Broaden Abstract Syntax Rules:

The `GraphASTAnalyzer` cleanly verifies nodes, but doesn't trace data models passed inside edges. To improve the *State Management Rigor* dimension, the AST visitor must be expanded to parse Pydantic schema keys. (Modify: `src/tools/repo_tools.py`)

3. (Low Priority) Centralize Configuration Logistics:

Right now variables like `.env` are scattered. To improve *Safe Tool Engineering*, create a unified singleton Settings class utilizing `pydantic-settings`. (Modify: `src/config.py` new file)