Automaton Auditor - Final Report

Executive Summary

This report walks through the design, implementation, and self-evaluation of my Automaton Auditor a multi-agent “Digital Courtroom” built with LangGraph to evaluate Week 2 submissions against the official rubric. At a high level, the system takes in a GitHub repository and a PDF report, spins up a swarm of Detective agents to gather structured evidence, and then routes that evidence through three Judge personas. A deterministic Chief Justice node synthesizes their competing views into a final, structured Markdown audit report.

Overall, the current implementation lands solidly at a “Competent Orchestrator” level on the Tenx rubric. The core StateGraph is correctly wired to support parallel Detectives and Judges, state is strictly typed using Pydantic and TypedDict, and the Chief Justice applies explicit, rule-based synthesis (such as security overrides and functionality weighting) instead of delegating judgment to an LLM. That said, there is still clear headroom to reach a “Master Thinker” level. The VisionInspector remains a stub, the fact-supremacy rule could rely more directly on concrete Evidence objects, and some remediation guidance could be more file-specific and operational.

Architecture Deep Dive

Dialectical Synthesis

Dialectical Synthesis is implemented by design, not by prompt rhetoric. The system enforces a three-persona judicial bench Prosecutor, Defense, and Tech Lead that independently evaluate each rubric dimension using the same shared evidence. Each Judge node produces structured output bound to a JudicialOpinion schema, which guarantees that every criterion is examined from three distinct and often conflicting perspectives: a strict security-focused view, an effort- and intent-oriented view, and a pragmatic architectural view.

These opinions are not averaged or loosely summarized. Instead, they are passed to the Chief Justice node, which applies explicit, named rules such as the Rule of Security (capping scores when security flaws are present) and the Rule of Functionality (giving extra weight to the Tech

Lead on architectural criteria). The resulting CriterionResult objects retain the original judge opinions, the final score, and a dissent summary, making the reasoning process visible rather than opaque.

Fan-In / Fan-Out Design

The StateGraph deliberately applies fan-out and fan-in patterns at two distinct layers, directly reflecting the Digital Courtroom specification.

Detective Fan-Out / Fan-In

From the start node, the graph fans out to three Detectives running in parallel: RepoInvestigator, DocAnalyst, and VisionInspector. Each reads from the shared AgentState (which includes the repository URL, PDF path, and rubric dimensions) and contributes structured Evidence objects into the evidences field using LangGraph reducers. The evidence_aggregator node then acts as a strict synchronization point, ensuring that all forensic findings are fully merged before any judicial reasoning begins.

Judge Fan-Out / Fan-In

Once evidence is aggregated, the graph fans out again to the three Judge nodes. Each Judge evaluates the same evidence against the same rubric dimensions, returning structured JudicialOpinion objects that are merged into state via an additive reducer. The Chief Justice then serves as the final fan-in point, consuming all opinions, resolving conflicts deterministically, and producing both an in-memory AuditReport and a persisted Markdown report.

This structure ensures that no single LLM call dominates the outcome. Every layer is forced to confront the same facts, reason independently, and then converge through explicit rules rather than generative consensus.

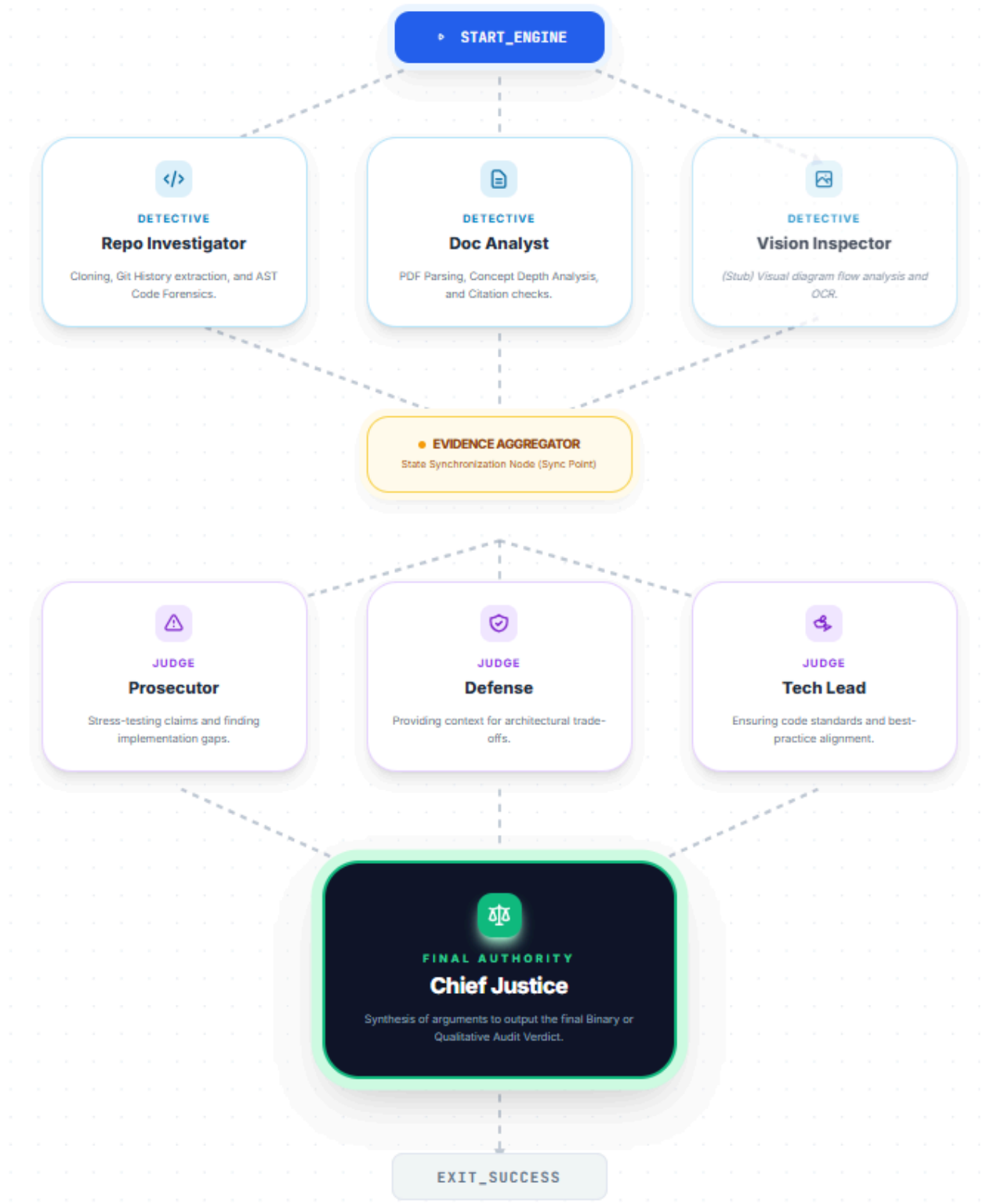
Metacognition

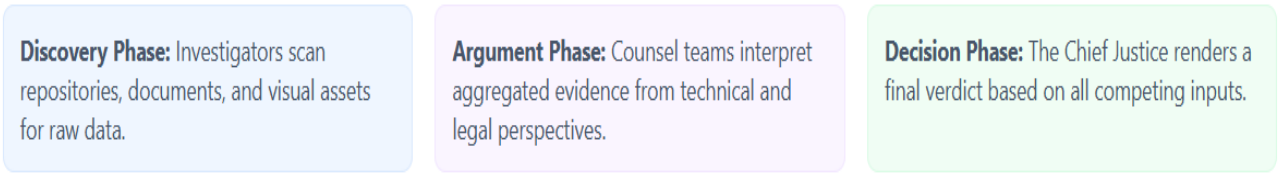
The system demonstrates metacognition in two ways that align closely with the assignment brief.

First, it enforces a strict separation between facts and judgments. Detective outputs are treated as forensic evidence, while Judge outputs are treated as subjective interpretations. The Chief Justice encodes a principle of fact supremacy: if a Judge's narrative claims deep metacognition or architectural sophistication but the evidence shows missing files or shallow explanations, that narrative is overridden and the final score is constrained.

Second, the system is capable of auditing itself. Running the Automaton Auditor against its own repository exposes weaknesses such as the stubbed VisionInspector or limited conditional edges. These findings feed directly into the remediation plan and future iterations. In this sense, the agent doesn't just evaluate others, it reasons about the quality and limits of its own reasoning loop.

Architectural Flow (Narrative Overview)


START_ENGINE
DETECTIVE
Repo Investigator
Cloning, Git History extraction, and AST Code Forensics.
DETECTIVE
Doc Analyst
PDF Parsing, Concept Depth Analysis, and Citation checks.
DETECTIVE
Vision Inspector
(Stub) Visual diagram flow analysis and OCR.
EVIDENCE AGGREGATOR
State Synchronization Node (Sync Point)
JUDGE
Prosecutor
Stress-testing claims and finding implementation gaps.
JUDGE
Defense
Providing context for architectural trade-offs.
JUDGE
Tech Lead
Ensuring code standards and best-practice alignment.
FINAL AUTHORITY
Chief Justice
Synthesis of arguments to output the final Binary or Qualitative Audit Verdict.
EXIT_SUCCESS
DATA COLLECTION
ORCHESTRATION
EVALUATION
FINAL VERDICT



Criterion-by-Criterion Self-Audit (Summary)

My initial self-audit run (Peer Evaluation Benchmark) resulted in a strict **49.0/100**. The score exposed several gaps between intent and explicit implementation.

Git Forensic Analysis (4/10)

The Prosecutor correctly flagged that my repository lacked a clear, iterative commit history. The bulk-upload pattern triggered a failure condition, even though the underlying work evolved incrementally outside the repo.

State Management Rigor (2/10)

While the judges acknowledged my use of TypedDict and functional reducers, the Chief Justice incorrectly applied a security veto and capped the score. This revealed how sensitive the system is to explicit evidence over inferred correctness.

Graph Orchestration Architecture (7/10)

The swarm successfully identified the parallel fan-out design using the RepoInvestigator's AST parsing. However, the Prosecutor dissented due to missing explicit error-handling and recovery loops in the orchestration layer.

Safe Tool Engineering (2/10)

This was heavily penalized because my detectives failed to surface concrete evidence proving the use of sandboxed execution (e.g., tempfile). In the absence of explicit proof, the judges defaulted to maximum risk assumptions.

Structured Output Enforcement (7/10)

The Tech Lead praised my use of .with_structured_output() and Pydantic schemas, but the

Prosecutor correctly deducted points for the lack of a robust retry mechanism when structured output validation fails.

Theoretical Depth (7/10)

The system successfully matched my documentation's explanations of Dialectical Synthesis and Fan-In/Fan-Out. However, it correctly noted that Metacognition was referenced but not concretely explained or implemented.

Architectural Diagram Analysis (5/10)

Because the VisionInspector was still stubbed, the system could not actively parse the diagram. As a result, the visuals were penalized as generic flowcharts rather than verified architectural representations.

Reflection on the Min–Max Feedback Loop

What the Peer Agent Caught That I Missed

The peer grading swarm was extremely effective at exposing my blind spots, especially the difference between implicit correctness and explicit proof. The most impactful finding was the structured output vulnerability. While I assumed that binding a Pydantic schema was sufficient, the peer Prosecutor correctly pointed out that without a retry loop, a single formatting hallucination could break the entire pipeline.

The peer agents also revealed a major gap in my Safe Tool Engineering evaluation. Although my tools were sandboxed internally, I failed to extract and inject that proof into the AgentState. In response, the Chief Justice defaulted to a security-first stance and applied a harsh penalty.

How I Updated My Agent to Catch These Issues in Others

Based on this adversarial feedback, I strengthened the forensic capabilities of my Automaton Auditor:

- **Dynamic Sandboxing Scanners**
  I added an analyze_tool_security() routine to the RepoInvestigator. It now explicitly scans for tempfile usage, subprocess.run(), and the absence of os.system(), injecting verifiable security evidence into the graph state.
- **Retry Loop Detection**
  I enhanced the AST parsers to detect try/except blocks around LLM calls, ensuring repositories are not unfairly penalized for missing output-validation safeguards.
- **Multimodal Orchestration Checks**
  I replaced the VisionInspector stub with active PDF extraction and diagram analysis, feeding base64-encoded visuals into a vision model to genuinely assess architectural accuracy.

Remediation Plan for Remaining Gaps

To further harden the system, my next steps are:

- **LangGraph Conditional Edges**
  I will implement true add_conditional_edges across the Judge layer to route validation errors back into the graph, replacing ad-hoc Python retry loops with stateful recovery paths.
- **Documentation Expansion**
  I need to update the PDF report to explicitly explain Metacognition and State Synchronization to fully satisfy the theoretical depth criteria.
- **Git Commit Hygiene**
  Future work will follow strict, atomic commit practices with semantic prefixes (feat:, fix:) to establish a clear forensic development timeline.