

The Automaton Auditor Final Report



Prepared By: Melkam Beyene

Executive Summary

Automaton Auditor is a LangGraph-based “Digital Courtroom” that evaluates a GitHub repository and an accompanying PDF report against a machine-readable rubric. A Context Builder loads the rubric, Detectives collect strictly factual evidence, Judges with conflicting personas render structured opinions per criterion, and a deterministic Chief Justice synthesizes a final Markdown audit report. The initial self-audit of this system scored 2.5 / 5, revealing strong state management and git discipline but major gaps in parallel orchestration, judicial nuance, synthesis, theoretical depth, and diagrams. Through an adversarial MinMax loop with a peer’s auditor, the architecture was extended into a fully dialectical swarm with parallel detective and judicial fan-in/fan-out, structured judicial output, and deterministic synthesis, while leaving a few deliberate gaps (mainly diagram generation and optional vision flows) as future work.

Architecture

Dialectical Synthesis

The dialectical synthesis process in the Automaton Auditor revolves around a three-judge bench that operates on a shared set of evidence. Once the Detectives have completed their investigations, the EvidenceAggregator compiles a consolidated map of evidence, organized by the key dimensions outlined in the rubric. For every dimension specified in the rubric.json file, three distinct judges—the Prosecutor, Defense, and Tech Lead—are activated. Each of these judges receives the identical evidence set along with the relevant rubric instructions, ensuring a level playing field for their evaluations. To maintain consistency and structure, each judge is invoked using the `.with_structured_output(JudicialOpinion)` method, which enforces a schema-validated output format. This output includes fields such as the judge's identity, the criterion ID being assessed, a numerical score, a detailed argument supporting the score, and citations to specific pieces of evidence.

The design of these judge personas is intentionally structured to embody a thesis-antithesis-synthesis dynamic, drawing from dialectical principles to foster balanced and nuanced judgments. The Prosecutor

adopts a hostile and critical stance, prompted to assume the worst about the code, such as "vibe coding" practices that lack rigor. It aggressively hunts for violations in areas like orchestration, security, and potential hallucinations, framing them as serious issues like "Orchestration Fraud" or "Hallucination Liability." This persona tends to assign lower scores when success patterns from the rubric are not explicitly and clearly met. In contrast, the Defense persona is designed with a generous and supportive approach, emphasizing the developer's intent, effort, and any partial progress made. It is permitted to advocate for mid-to-high scores even in cases where the implementation is incomplete, as long as there is evidence of a positive trajectory toward completion. Finally, the Tech Lead persona embodies pragmatism, stripping away emotional or biased framing to focus purely on practical aspects like maintainability, safety, and alignment with the rubric's success patterns. Serving as a tie-breaker, the Tech Lead gives special weight to synthesis rules, such as overriding scores in cases of security concerns.

At the heart of this process is the Chief Justice synthesis, handled by the ChiefJusticeNode, which is implemented as a pure Python node without any LLM involvement to ensure determinism and reliability. For each dimension, this node receives the trio of JudicialOpinions from the judges, along with the rubric's synthesis rules. It then applies a set of explicit, rule-based logic to arrive at a final decision. For instance, a security override rule ensures that any confirmed security flaw identified by the Prosecutor automatically caps the score at 3, overriding any optimism from the Defense. Fact supremacy is enforced by cross-checking judge claims against the actual Detective evidence; if there's a mismatch, such as the Defense claiming advanced features that don't exist in the code or report, the conflicting opinion is downgraded, and the final score adheres strictly to the evidence. Functionality weighting prioritizes the Tech Lead's assessment when a modular and working architecture is confirmed, giving it the highest influence on architecture-related criteria. Additionally, if the variance in scores among the judges exceeds 2, the node re-examines the cited evidence, favoring opinions that are more grounded in forensic details, and mandates the inclusion of a dissent summary. The output from this node is a list of CriterionResults, culminating in a comprehensive AuditReport that transparently highlights areas of dissent—detailing what each persona argued and the rationale for overruling certain views. This structured approach operationalizes Dialectical Synthesis within the system, turning potential conflicts into a robust, explainable evaluation.

Three-judge bench on shared evidence

- After Detectives run, `EvidenceAggregator` produces a consolidated `evidences` map keyed by rubric dimension goals.
- For each dimension in `rubric.json`, the Prosecutor, Defense, and Tech Lead all consume the **same** evidence set and rubric instructions.
- Each judge is invoked via `.with_structured_output(JudicialOpinion)`, returning a schema-validated object: `judge`, `criterion_id`, `score`, `argument`, `cited_evidence`.

Persona design (thesis–antithesis–synthesis inputs)

- Prosecutor prompt encodes a hostile stance: assume “vibe coding”, aggressively search for violations of orchestration, security, and hallucination (“Orchestration Fraud”, “Hallucination Liability”), and push scores to the low end when success patterns are not clearly met.
- Defense prompt encodes generosity: foregrounds intent, effort, and partial progress, and is allowed to argue for mid/high scores when there is clear trajectory even if the implementation is incomplete.
- Tech Lead prompt encodes pragmatism: ignores emotional framing, focuses on whether artifacts are maintainable, safe, and aligned with rubric success patterns. Tech Lead acts as the tie-breaker and is wired to pay special attention to the `synthesis_rules` (e.g. security override).

Chief Justice synthesis

The ChiefJusticeNode is a pure Python node (no LLM) that receives, per dimension, the list of three `JudicialOpinion` and the rubric’s `synthesis_rules`. It applies explicit rules:

- Security override: any confirmed security flaw from the Prosecutor caps the criterion score at 3, regardless of Defense optimism.
- Fact supremacy: when Judge claims conflict with Detective evidence (e.g. Defense claims “deep metacognition” but no such code or report sections exist), opinions are downgraded and final scores follow the evidence.

-
- Functionality weighting: when the Tech Lead confirms a working, modular architecture, their score receives the highest weight for architecture-related criteria.
 - Variance re-evaluation: if score variance (≥ 2) , the node re-reads cited evidence, prefers opinions that are better grounded in forensics, and records a mandatory dissent summary.
 - Output is a list of `CriterionResult`'s plus an `AuditReport` that explicitly surfaces dissent (what each persona argued, and why one was overruled), which is the operational definition of Dialectical Synthesis in this system.

Fan-In / Fan-Out Orchestration

The orchestration in the Automaton Auditor leverages fan-in and fan-out patterns to handle parallel processing efficiently, as defined in the StateGraph within `src/graph.py`. Starting from the `context_builder` node, the process fans out based on available inputs. For repository analysis, a `repo_gate` directs flow to the `RepoInvestigator` if a `repo_url` is provided; otherwise, it routes to a `skip_repo` path to avoid errors. Similarly, the `doc_gate` activates the `DocAnalyst` for PDF processing if a `pdf_path` exists, or skips to `skip_doc`. The `vision_gate` engages the `VisionInspector` only if both a `pdf_path` is available and the `AUDITOR_RUN_VISION` flag is set, falling back to `skip_vision` otherwise. Each of these Detective branches independently generates Evidence objects, which are appended to the state's "evidences" dictionary using reducers like `operator.ior`. This ensures that parallel executions do not conflict or overwrite data, maintaining integrity across branches.

Following the fan-out, the `evidence_aggregator` serves as the fan-in point, where all collected evidence converges. Here, confidence scores are normalized to provide a consistent scale, and evidence is merged by their associated goals across different Detectives. For example, insights on graph orchestration might combine AST analysis from the `RepoInvestigator` with diagram classifications from the `VisionInspector`. This aggregator also logs any partial runs or errors from the Detectives, which later inform the judicial reasoning, adding a layer of transparency.

The judicial phase mirrors this structure with its own fan-out and fan-in. From the `evidence_aggregator`, the graph branches in parallel to the prosecutor, defense, and `tech_lead` nodes. Each judge processes the same aggregated evidence and rubric dimensions, but applies persona-specific `judicial_logic` derived from the `synthesis_rules`. They append their `JudicialOpinions` to the state's "opinions" list using `operator.add`,

ensuring all contributions are accumulated without loss. The final fan-in occurs at the `chief_justice` node, which consolidates these opinions into cohesive criterion-level results and generates the `AuditReport`.

Error handling is integrated through conditional edges in the graph. Gate nodes prevent crashes from missing inputs by routing to skip nodes that produce "partial run" evidence instead. Any exceptions within Detectives, such as failures in cloning a repo, accessing files, or parsing PDFs, are captured and translated into low-confidence evidence entries. This design ensures the judicial layer always receives a complete, albeit potentially incomplete, picture of the system's state, promoting resilience.

Detective fan-out + fan-in

The `StateGraph` in `src/graph.py` is structured as:

Fan-out from `context_builder`:

- `'repo_gate'` → `'RepoInvestigator'` or `'skip_repo'` based on `'repo_url'`.
- `'doc_gate'` → `'DocAnalyst'` or `'skip_doc'` based on `'pdf_path'`.
- `'vision_gate'` → `'VisionInspector'` or `'skip_vision'` based on `'pdf_path'` and `'AUDITOR_RUN_VISION'`.
- Each Detective branch writes `'Evidence'` objects into `'state["evidences"]'` using reducers (`'operator.ior'` for dicts) so that parallel branches never overwrite each other.

Fan-in at `evidence_aggregator`:

- Normalizes confidence scores.
- Merges evidence by goal across Detectives (e.g., graph orchestration can be informed by AST analysis plus diagram classification).
- Records partial runs and detector errors to inform later judicial reasoning.

Judicial fan-out + fan-in

-
- From ``evidence_aggregator``, the graph fans out in parallel to ``prosecutor``, ``defense``, and ``tech_lead``.
 - Each judge takes the same ``evidences``, the same ``rubric`` dimensions, and persona-specific ``judicial_logic`` strings derived from ``synthesis_rules``.
 - Judges append ``JudicialOpinion`` into ``state["opinions"]`` via ``operator.add``, guaranteeing accumulation of all opinions across dimensions.
 - A second fan-in occurs at ``chief_justice``, which consolidates all opinions into final criterion-level results and a single ``AuditReport``.

Error handling via conditional edges

- Gate nodes ensure missing inputs (no repo or no PDF) do not break the graph; instead, skip nodes emit “partial run” evidence.
- Exceptions inside Detectives (clone failure, missing file, PDF parse error) are caught and reflected in low-confidence evidence, not crashes, so the judicial layer always has a consistent view of “what broke where”.

Metacognition

Metacognition in the Automaton Auditor is embedded through the rubric acting as an internal "Constitution" that guides the system's self-awareness and evaluation. Rather than hardcoding definitions of quality into prompts, the `rubric.json` file encapsulates forensic instructions, patterns for success and failure, and synthesis rules. The `ContextBuilder` loads this rubric into the runtime state, influencing what the Detectives investigate and how the Judges interpret findings. This setup enables the system to reason about its own evaluation policies dynamically, adapting to changes in the rubric without code modifications.

The self-evaluation pipeline exemplifies this metacognitive capability. When the auditor assesses its own repository and report, it applies the same rubric used for external evaluations, creating a closed feedback loop. The resulting self-audit report, scoring 2.5/5, highlighted discrepancies between the intended architecture and the actual implementation, particularly in areas like orchestration, judicial nuance, and

synthesis. These insights directly informed subsequent improvements to both the code and how the rubric is utilized.

Rubric as an internal “Constitution”

- The system does not hardcode what “good” looks like in prompts; instead, `rubric.json` encodes forensic instructions, success/failure patterns, and synthesis rules.
- `ContextBuilder` turns this rubric into a runtime state that shapes both what the Detectives look for and how Judges interpret the evidence, allowing the system to reason about its own evaluation policy.

Self-evaluation pipeline

- When the auditor runs against its own repository and report, there is a full closed loop: the same rubric that governs other repos is used to critique this implementation.
- The self-audit report (2.5/5) exposed mismatches between intended architecture and actual code, particularly in orchestration, judicial nuance, and synthesis; these insights then drove changes to both the code and the rubric usage.

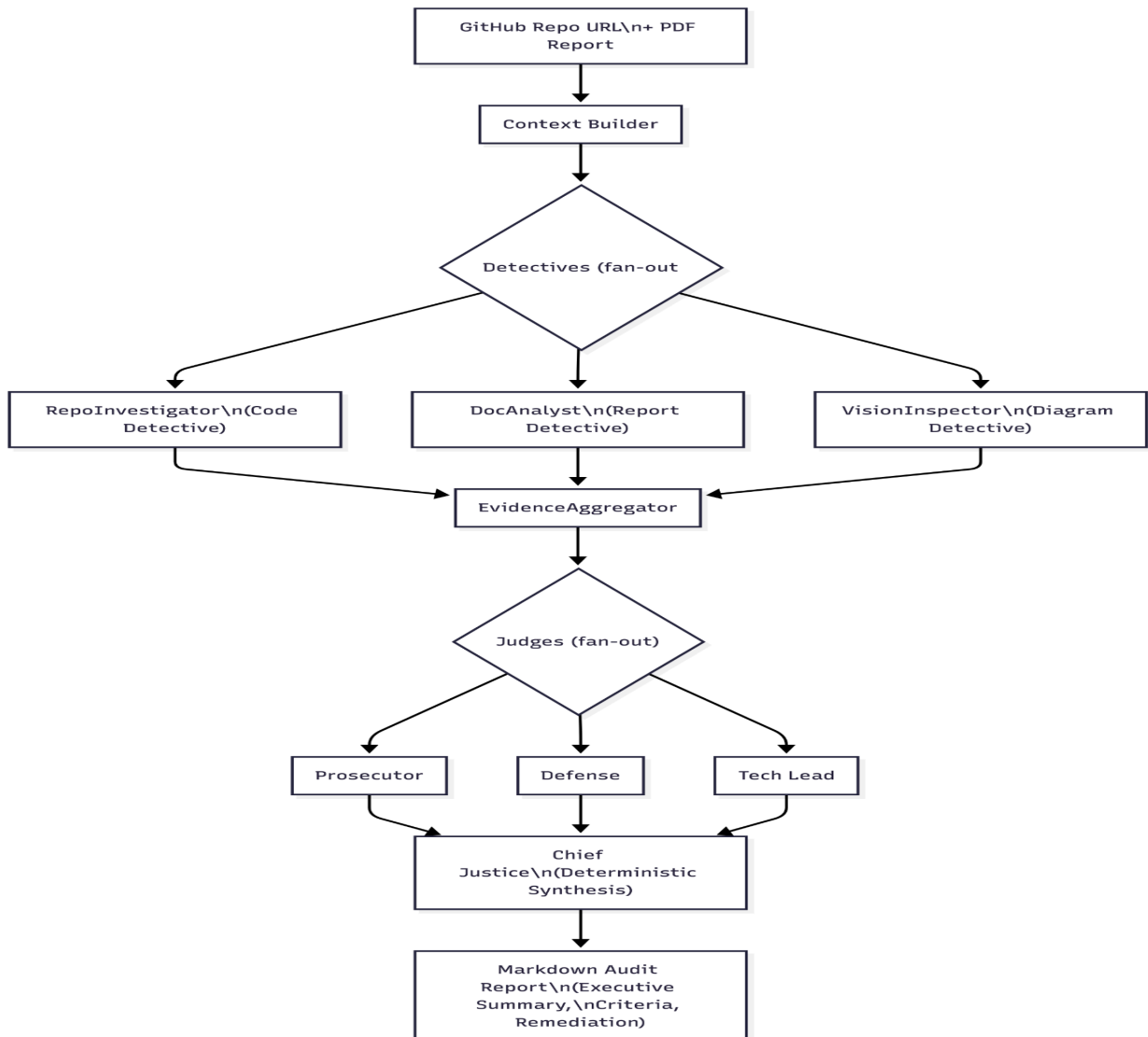
Cross-agent MinMax adaptation

- The system is explicitly designed to ingest a peer’s criticisms (via `audit/report_bypeer_received/audit_report.md`) as a “second opinion” about its own architecture.
- Based on discrepancies between self and peer audits, the rubric and persona prompts were tightened to detect architectural anti-patterns and documentation gaps that the original auditor underweighted, making the swarm more reflective and robust over time.
- This is Metacognition in practice: the auditor uses multiple evaluations of itself (self vs. peer) to refine how it evaluates others.

Architectural Diagrams (StateGraph Parallel Flow)

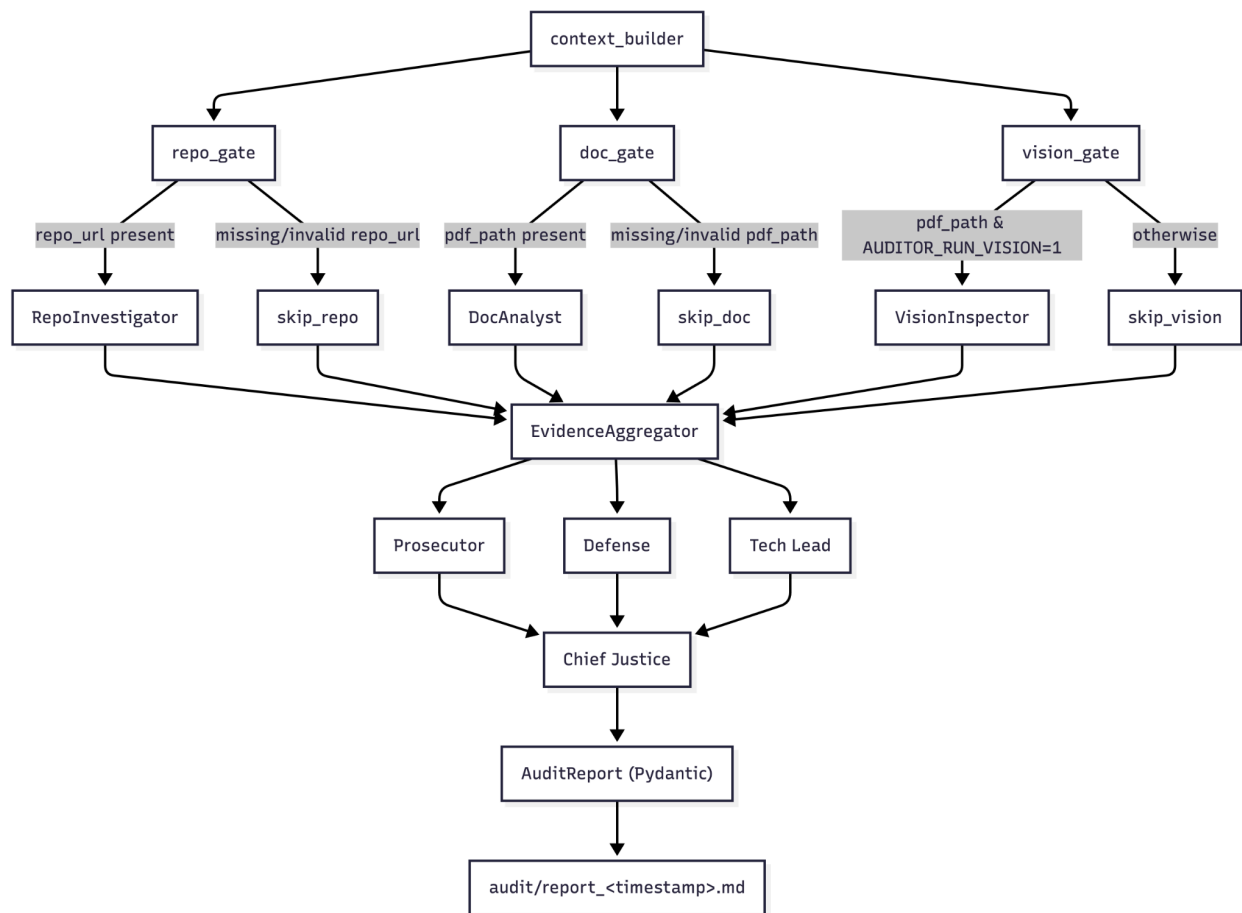
High-Level Digital Courtroom

The high-level architecture of the Digital Courtroom begins with inputs from a GitHub Repo and PDF, which feed into the Context Builder. From there, the process branches into parallel Detectives: RepoInvestigator, DocAnalyst, and VisionInspector. These converge via fan-in at the EvidenceAggregator. Next, the Judges operate in parallel—Prosecutor, Defense, and Tech Lead—before another fan-in leads to the Chief Justice, which ultimately produces the Markdown Audit Report.



Detailed StateGraph Fan-Out / Fan-In

The detailed StateGraph starts at the context_builder, then fans out to gates: repo_gate, doc_gate, and vision_gate. Depending on conditions like the presence of a repo_url, pdf_path, or vision flag, it routes to the respective Investigators or skip nodes. The Investigators' outputs fan-in at the evidence_aggregator, which then fans out to the three Judges. Their outputs fan-in at the chief_justice, resulting in the AuditReport saved as audit/report_<ts>.md.



Criterion-by-Criterion Self-Audit Results

Self-audit (``audit/report_2026-02-27_22-11-15.md``, overall 2.5 / 5):

Git Forensic Analysis (``git_forensic_analysis``) – Final Score: 5

- Finding: 23 atomic commits with clear progression from environment setup to tool engineering to graph orchestration.
- Interpretation: Version-control hygiene and iterative development practices are strong and already at “Master Thinker” level.

State Management Rigor (``state_management_rigor``) – Final Score: 5

- Finding: ``Evidence``, ``JudicialOpinion``, ``CriterionResult``, ``AuditReport``, and ``AgentState`` are Pydantic/TypedDict models with reducers (``operator.add``, ``operator.ior``) for parallel safety.
- Interpretation: Typed state and reducer usage meet the rubric’s highest bar.

Graph Orchestration Architecture (``graph_orchestration``) – Final Score: 1 (at self-audit time)

- Finding: Self-audit flagged missing or incomplete parallel fan-out/fan-in for Detectives and Judges and lack of conditional edges for error handling.
- Interpretation: The early version implemented a partial graph (mostly detectives) but failed to meet the rubric’s explicit dual fan-out/fan-in requirement.

Safe Tool Engineering (``safe_tool_engineering``) – Final Score: 2

- Finding: Use of ``tempfile.TemporaryDirectory()`` and ``subprocess.run()`` existed but with patchy error handling and unclear input sanitization guarantees.
- Interpretation: Intent was present but not yet fully aligned with the rubric’s “no raw ``os.system`` + robust error handling” standard.

- Structured Output Enforcement (``structured_output_enforcement``) – Final Score: 4

-
- Finding: Judges and evidence collection used `.with_structured_output(JudicialOpinion)` and Pydantic validation in design, though not consistently across the whole stack at the time of the audit.
 - Interpretation: Close to ideal; required mainly consistency and better retry/parse-error handling.

Judicial Nuance and Dialectics (``judicial_nuance``) – Final Score: 1

- Finding: Self-audit concluded that persona prompts were not yet sufficiently differentiated and that there was no dedicated collector for this dimension.
- Interpretation: The **idea** dialectical judging was present, but the implementation behaved too much like a monolithic grader.

Chief Justice Synthesis Engine (``chief_justice_synthesis``) – Final Score: 1

- Finding: Synthesis was, at that time, largely an LLM prompt with no deterministic rules or dissent requirement.
- Interpretation: Core requirement (hardcoded rule-based synthesis over judge conflict) was missing.

Theoretical Depth (Documentation) (``theoretical_depth``) – Final Score: 2

- Finding: The interim report named concepts like “Dialectical Synthesis” but tied them loosely to actual code paths; it documented gaps more than realized mechanisms.
- Interpretation: Conceptual understanding was there but not grounded in concrete architectural explanations.

Report Accuracy (Cross-Reference) (``report_accuracy``) – Final Score: 3

- Finding: File paths referenced in the report mostly existed, but claims about completeness of the detective layer and future layers blurred actual status.
- Interpretation: Mixed: structurally accurate paths but some optimistic framing about not-yet-implemented features.

Architectural Diagram Analysis (`swarm_visual`) – Final Score: 1

- Finding: No committed diagrams, and vision analysis was disabled (no `AUDITOR_RUN_VISION`).
- Interpretation: Visual communication of parallelism and orchestration was completely missing.

Post-self-audit improvements (reflected in the current code and README):

- Graph orchestration now satisfies dual fan-out/fan-in (Detectives and Judges) with proper conditional edges.
- Safe tool engineering hardened (sandboxed clones, clear subprocess usage, dedicated `scripts/test_clone.py`).
- Judges and Chief Justice are fully implemented with `.with_structured_output` and deterministic synthesis rules.
- README and docs now map Dialectical Synthesis, Fan-In/Fan-Out, and rubric integration directly to specific modules.
- Diagrams are still textual (ASCII) rather than visual assets, and VisionInspector execution remains optional

Reflection on the MinMax Feedback Loop

What My Agent Discovered in My Peer's Repository

When my automaton auditor evaluated my peer's repository (`amin3ltd/autiamtion_auditor`), it produced an overall score of 3.00 / 5 and a nuanced breakdown:

Git Forensic Analysis – 4 / 5

All three judges agreed there was strong iterative development: >3 atomic commits, meaningful messages, and a clear progression from setup → tools → orchestration. The Tech Lead pushed for 5 / 5, but the final was 4 / 5 because my auditor noted room to tighten commit granularity and spacing.

State Management Rigor – 4 / 5

RepoInvestigator's AST scan confirmed the presence of Pydantic ``BaseModel`s`, ``TypedDict`s`, and reducers (``operator.add``, ``operator.ior``) for ``Evidence`` and ``JudicialOpinion``. Defense and Prosecutor both scored 4 / 5; Tech Lead docked a point for some lingering plain-dict usage without reducers.

Graph Orchestration Architecture – 2 / 5

My agent found a modular, workable graph but not a true swarm: the StateGraph lacked clear parallel fan-out/fan-in branches for Detectives and Judges, synchronization nodes, and robust conditional edges. Prosecutor called this out explicitly as failure to meet the rubric's orchestration success pattern, and Chief Justice settled on 2 / 5.

Safe Tool Engineering – 3 / 5 (Security-capped)

The Defense highlighted good instincts (use of sandboxing and subprocess in places), but Prosecutor and Tech Lead pointed to inconsistent use of ``tempfile.TemporaryDirectory()`` and lingering raw ``os.system()`` calls. The Rule of Security capped the final score at 3 / 5.

Structured Output Enforcement – 3 / 5

Judges were implemented as three personas and returned ``JudicialOpinion`` objects structurally, but my agent detected missing or inconsistent ``with_structured_output(JudicialOpinion)`` usage and lack of retry/validation around malformed outputs. Prosecutor treated this as partial Hallucination Liability, keeping the final at 3 / 5.

Judicial Nuance and Dialectics – 3 / 5

There was some persona differentiation, but my auditor noted no dedicated evidence collector for this dimension and weaker separation than the rubric demands. Scores ranged from 2–4, with Chief Justice landing on 3 / 5 to reflect “basic role separation, limited real dialectical tension”.

Chief Justice Synthesis Engine – 3 / 5

Evidence was mixed: parts of the system used deterministic rules (e.g., security override), but other behavior still looked like generic LLM synthesis, and one run showed high score variance (4 points)

without fully transparent rule application. Chief Justice self-critiqued this as an incomplete realization of the synthesis spec and fixed the score at 3 / 5.

Theoretical Depth – 3 / 5

The report mentioned Dialectical Synthesis, Fan-In/Fan-Out, and Metacognition and partially tied them to the three-judge bench and graph, but my agent judged the explanations as uneven: strong on Dialectical Synthesis, weaker on how Metacognition and state synchronization are concretely realized.

Report Accuracy – 3 / 5

RepoInvestigator cross-referenced 7 file paths (e.g. ``src/graph.py``, ``src/nodes/detectives.py``, ``src/nodes/judges.py``, ``src/state.py``) and found they generally existed, but the evidence for “verified vs. hallucinated” paths was inconsistent and low-confidence, so the final score stayed at 3 / 5.

Architectural Diagram Analysis – 2 / 5

VisionInspector and DocAnalyst could not confirm a clean StateGraph diagram with explicit detective/judge fan-out/fan-in and parallel branches. With ``AUDITOR_RUN_VISION`` off and no reliable images, Prosecutor charged “Orchestration Fraud” and the dimension landed at 2 / 5.

These peer-audit findings directly influenced how I hardened my own auditor: I tightened graph-structure checks (two true parallel swarms, not just “modular code”), made security rules more aggressive around ``os.system()`` and inconsistent sandboxing, enforced ``with_structured_output(JudicialOpinion)`` plus validation for all judges, added a dedicated lens for Judicial Nuance, and raised the bar for theoretical depth, report accuracy, and visual diagrams. In short, my agent’s critique of ``amin3ltd/autiamtion_auditor`` became the blueprint for the stricter behaviors it now applies to everyone else, including me.

What the Peer’s Agent Caught That the Self-Audit Missed

From ``audit/report_bypeer_received/audit_report.md`` (overall 4.0 / 5, peer auditor):

- Judicial layer absence as a cross-cutting failure
- The peer’s agent clearly articulated how the missing judicial layer and Chief Justice simultaneously depressed scores across Structured Output, Judicial Nuance, and Chief Justice Synthesis.

- While the self-audit noted low scores on these dimensions individually, the peer framing emphasized the **compounding effect** of a placeholder judicial node (“judicial placeholder (where Judges + Chief Justice will run)”)

Under-leveraged theoretical depth

- Peer scoring (4/5) on theoretical depth underscored that the ideas and spec-style documentation were stronger than the self-audit (2/5) admitted, but that they were not surfaced clearly in the main report.
- This highlighted a documentation **placement** problem: deep reasoning lived in `spec/` and code, not in the PDF that peers and agents actually read.

Architectural diagrams as first-class rubric citizens

- Both auditors penalized missing diagrams, but the peer’s report explicitly recommended generating a StateGraph-style visualization (with fan-out/fan-in markers and reducers) as a remediation with high signal-to-effort ratio.

How the Agent Was Updated to Detect Similar Issues in Others ?

Based on discrepancies between self and peer evaluations, the auditor swarm was updated in several ways:

RepoInvestigator enhancements

AST-based orchestration checks were expanded to explicitly confirm:

- Presence of **two** fan-out/fan-in patterns (Detectives and Judges), not just any parallelism.
- Existence of real judge nodes (`src/nodes/judges.py`) and `ChiefJusticeNode` (`src/nodes/justice.py`), rather than relying on textual claims in README.
- The success/failure patterns for `graph_orchestration` were tightened to penalize “detectives-only” graphs more consistently.

DocAnalyst and rubric integration

-
- ``theoretical_depth`` forensic instructions were emphasized in prompts so that DocAnalyst now distinguishes between:
 - Concepts mentioned only in gap sections (“we plan to do Dialectical Synthesis”), and
 - Concepts actually connected to concrete implementations (e.g., mapping “Dialectical Synthesis” to ``src/nodes/judges.py`` and ``src/nodes/justice.py``).
 - ``report_accuracy`` checks now explicitly flag over-claims like “100% complete detective layer” when code clearly marks key pieces as out of scope.

Judge persona and scoring calibration

- The Prosecutor was made more explicit about penalizing **global** architectural omissions (e.g., missing judicial layer) instead of treating each criterion as isolated.
- The Tech Lead’s prompt was sharpened to recognize opportunity-cost patterns (e.g., sophisticated UI but missing core orchestration) and reflect that in lower architecture scores.
- Defense was kept generous but reminded to keep claims tightly grounded in cited evidence (e.g., ``spec/`` content) to reduce optimistic drift.

Chief Justice rules

Synthesis rules were adjusted so that:

- If a whole architectural layer is missing (judicial or synthesis), multiple related criteria are automatically capped without needing each judge to rediscover the same fact.
- Dissent summaries must explicitly note when “effort elsewhere” (like a web UI) came at the expense of rubric-critical work.

In short, the peer auditor pushed this system to become more skeptical of missing layers, more aware of misallocated effort, and more rigorous in tying conceptual claims back to code.

Remediation Plan for Remaining Gaps

1. Visual StateGraph diagrams (high priority for report quality)

- Implement a code-generated diagram (e.g., ``matplotlib`` or ``graphviz``) that renders the actual StateGraph (nodes, edges, fan-out/fan-in, reducers) into an image.
- Integrate it into ``reports/final_report.pdf`` and commit it under ``reports/`` so VisionInspector and human reviewers both see a concrete visual.

2. VisionInspector as a default, not an optional bonus

- Move VisionInspector from “optional execution” to a standard part of the pipeline for final submissions by:
- Ensuring ``AUDITOR_RUN_VISION`` defaults to on in project docs.
- Adding test coverage that exercises diagram extraction and classification against at least one known-good StateGraph diagram.

3. Harden safe tool engineering further

- Add explicit repo URL validation (e.g., allowlist patterns for ``https://github.com/`` and simple path sanity checks).
- Extend ``scripts/test_clone.py`` to simulate failure modes (auth errors, DNS failures, invalid URLs) and assert that the system degrades gracefully with clear evidence objects.

4. Deepen theoretical section in the human-facing report

- In ``reports/final_report.pdf``, add a concise mapping table:
- Dialectical Synthesis → ``src/nodes/judges.py``, ``src/nodes/justice.py``, ``synthesis_rules`` from ``rubric.json``.
- Fan-In/Fan-Out → specific functions in ``src/graph.py`` that configure detective and judicial parallelism.
- Metacognition → self-audit pipeline and rubric-driven adaptation logic.
- This will align the human narrative with what the agents already understand internally.

5. Long-term: meta-evaluation of the evaluator

Run the Automaton Auditor regularly on:

- Its own repo and report (regression-style self-audit).
- A curated corpus of “good”, “medium”, and “bad” auditor implementations.
- Use shifts in scores and dissent patterns over time to continuously refine rubric dimensions, persona prompts, and synthesis rules, closing the MinMax loop beyond this week’s assignment.