Interim Submission

The Automaton Auditor

Name Bethel Yohannes

Date Feb 25 206

Table of content

The Automaton Auditor

1 Architecture Decision Rationale

This system is designed as a multi-layered LangGraph workflow that evaluates a repository using structured evidence collection, adversarial judicial reasoning, and deterministic synthesis.

The architecture is divided into three layers:

Layer 1 – Detectives (Evidence Collection)
Layer 2 – Judges (Adversarial Evaluation)
Layer 3 – Chief Justice (Deterministic Synthesis Engine)

Below are the key architectural decisions made so far.

Pydantic structured models — Evidence, JudicialOpinion, CriterionResult, and AuditReport is choosed instead of raw dictionaries to ensure robustness, traceability, and long-term maintainability of the system.

1. Strong Type Safety and Structural Integrity

Each model explicitly defines its fields, data types, and constraints. For example:

- score: int = Field(ge=1, le=5) enforces that scores must remain within the rubric's defined bounds.
- Required fields are clearly separated from optional ones.
- Nested models enforce consistent hierarchical structure.

Unlike raw dictionaries, which accept arbitrary keys and values, Pydantic guarantees that every object adheres to a predefined schema. This eliminates ambiguity in agent communication and prevents malformed state updates from propagating through the system.

In a multi-agent environment, this structural discipline is especially important because outputs from one agent become inputs to another. Without strict schemas, subtle inconsistencies can compound into systemic failures.

2. Validation as a First-Class Guarantee

Validation is not treated as an afterthought — it is enforced at object creation.

If a judge attempts to produce:

- An invalid score (e.g., 7 instead of a 1–5 value),
- A missing required field,
- A wrongly typed attribute,
- A structurally incorrect nested object,

Pydantic raises an immediate validation error.

This provides:

- Automatic rubric enforcement
- Early failure detection
- Elimination of silent data corruption
- Higher trust in downstream decisions

For a judicial-style system where fairness and consistency matter, silent errors are unacceptable. Validation ensures every opinion complies with defined evaluation standards.

3. Self-Documenting and Auditable Schema

Each model functions as living documentation.

For example:

- `Evidence` clearly defines what constitutes admissible input.
- `JudicialOpinion` specifies exactly what a judge must return.
- `CriterionResult` formalizes scoring structure.

- AuditReport defines what the final evaluation must include.

This has several advantages:

- New contributors can understand the system by reading the models.
- Peer reviewers can audit structural expectations directly.
- Schema definitions become part of system governance.

Instead of relying on informal conventions, the schema becomes the authoritative contract of the system.

In systems designed for evaluation, review, or arbitration, clarity of structure is as important as correctness.

4. Production-Grade Extensibility

Structured models future-proof the architecture.

Because Pydantic models:

- Serialize cleanly to JSON
- Integrate naturally with APIs (e.g., FastAPI)
- Map cleanly to database schemas
- Support schema generation (OpenAPI / JSON Schema)
- Enable typed LLM output enforcement

The system can evolve from internal orchestration to:

- External API exposure
- Persistent audit logs
- Regulatory reporting
- Typed structured LLM responses
- Long-term storage and retrieval

Raw dictionaries would require additional validation layers and schema enforcement logic later. By designing correctly at the data layer, extensibility is built in from the start.

5. Determinism and System Trust

Judicial-style systems require:

- Predictable behavior
- Consistent structure
- Non-ambiguous scoring
- Traceable decision paths

Pydantic enforces correctness at the data layer — the foundation of the entire pipeline. When the state is guaranteed to be structurally sound, higher-level reasoning remains reliable.

1.2 Why AST Parsing Instead of Regex or Heuristics

The RepoInvestigator relies on Python's Abstract Syntax Tree (AST) parsing to analyze repository structure at a semantic level rather than treating source files as plain text.

Why Not Regex?

Regular expressions operate purely at the string level. While they can match patterns, they cannot understand program structure. This makes them unreliable for:

- Deeply nested functions
- Class hierarchies and inheritance chains
- Decorator patterns
- Multi-line definitions
- Context-sensitive constructs
- Edge cases involving comments or formatting

For example, a regex might detect `def foo(`, but it cannot reliably determine:

- Whether it is nested inside another function

- Whether it is part of a class
- Whether it is conditionally defined
- Whether it is syntactically valid

As codebases grow in complexity, regex-based heuristics break down. They introduce false positives, false negatives, and fragile parsing behavior. In an auditing system, this unreliability undermines evidence quality.

Why AST?

AST parsing transforms source code into a structured tree representation that reflects the actual grammar of the language.

This enables:

- Precise function and class detection
- Import graph inspection
- Call graph analysis
- Decorator awareness
- Node-level traversal of syntax elements
- Static detection of unsafe calls (e.g., os.system, subprocess.Popen)

Because the AST is derived from valid Python syntax, analysis becomes:

- Deterministic
- Reproducible
- Structurally grounded
- Resistant to formatting tricks
- Less hallucination-prone

Most importantly, it ensures that evidence is derived from semantic structure — not textual approximation.

The Prosecutor and Tech Lead therefore evaluate real program structure, not string patterns that merely resemble code.

In a judicial-style evaluation pipeline, evidence integrity is foundational. AST parsing guarantees that structural claims correspond to actual executable syntax.

1.3 Sandboxing Strategy

Since the system analyzes arbitrary public repositories, security is treated as a primary architectural constraint.

Controlled Repository Cloning

Repositories are cloned using:

- tempfile-based ephemeral directories
- Local, isolated storage
- Non-persistent working environments

No repository is cloned into privileged system paths, and all clones are disposable after analysis.

Strict Non-Execution Policy(No code execution)

The system performs static analysis only. It does not:

- Execute repository code
- Import dynamic modules
- Run setup scripts
- Trigger arbitrary shell commands derived from repo content

Key protections include:

- No direct shell execution of user-controlled input
- No dynamic imports of cloned modules
- Read-only static inspection

- Isolated analysis directory

This eliminates an entire class of attack vectors.

Threat Model Coverage

The sandboxing strategy mitigates:

- Malicious repositories embedding harmful scripts
- Path traversal attacks
- Dependency confusion risks
- Remote code execution
- Environment poisoning

Security is especially critical because the system operates on untrusted input. By separating analysis from execution and enforcing isolation, the architecture minimizes attack surface while preserving analytical depth.

Reducer-Based State Management

The LangGraph state management design uses reducer functions to ensure safe aggregation of parallel node outputs.

Specifically:

- `operator.ior` merges evidence dictionaries
- `operator.add` accumulates judicial opinions

Why Reducers?

In a fan-out/fan-in graph architecture, multiple nodes execute concurrently and write to shared state. Without disciplined merge logic, nodes can overwrite each other’s outputs.

Reducers ensure:

- Evidence is merged, not replaced
- Judicial opinions are appended, not overridden
- Parallel node outputs are preserved
- State transitions remain predictable

This avoids race-condition overwrites and guarantees compositional integrity.

Concurrency Safety

The reducer pattern ensures that:

- Detectives can emit evidence independently

- Judges can submit opinions without conflict
- Aggregation is deterministic

This design is essential for scalable orchestration. As additional evaluators or detectors are introduced, the merge logic remains stable and mathematically well-defined.

In short, reducers formalize how knowledge accumulates in the system.

2Known Gaps and Forward Plan

At the time of this interim submission:

- Layer 1 (Detectives) is implemented.
- Partial graph wiring exists.
- Judicial models are defined.
- Synthesis logic is scaffolded but not fully operational.

The following outlines the structured gap analysis and forward plan.

2.1 Missing Full Judicial Reasoning Implementation

Current State

- JudicialOpinion model is defined.
- Judge personas are conceptually specified.
- Evaluation prompts are not yet fully implemented.
- Structured LLM output enforcement is not wired.

Forward Plan

Implement three judicial nodes:

1. **Prosecutor**
    - Security-first evaluation
    - Vulnerability detection emphasis
    - Risk severity assessment
2. **Defense**
    - Effort recognition
    - Intent interpretation
    - Documentation weighting
3. **TechLead**
    - Architectural feasibility
    - Scalability evaluation
    - Maintainability assessment

Each judge will:

- Evaluate all rubric dimensions
- Produce structured JudicialOpinion output
- Bind LLM responses to strict Pydantic schema validation

This enforces uniform scoring structure and prevents malformed judicial output.

Milestone: Next implementation phase.

2.2 Chief Justice Synthesis Rules Not Fully Implemented

Current State

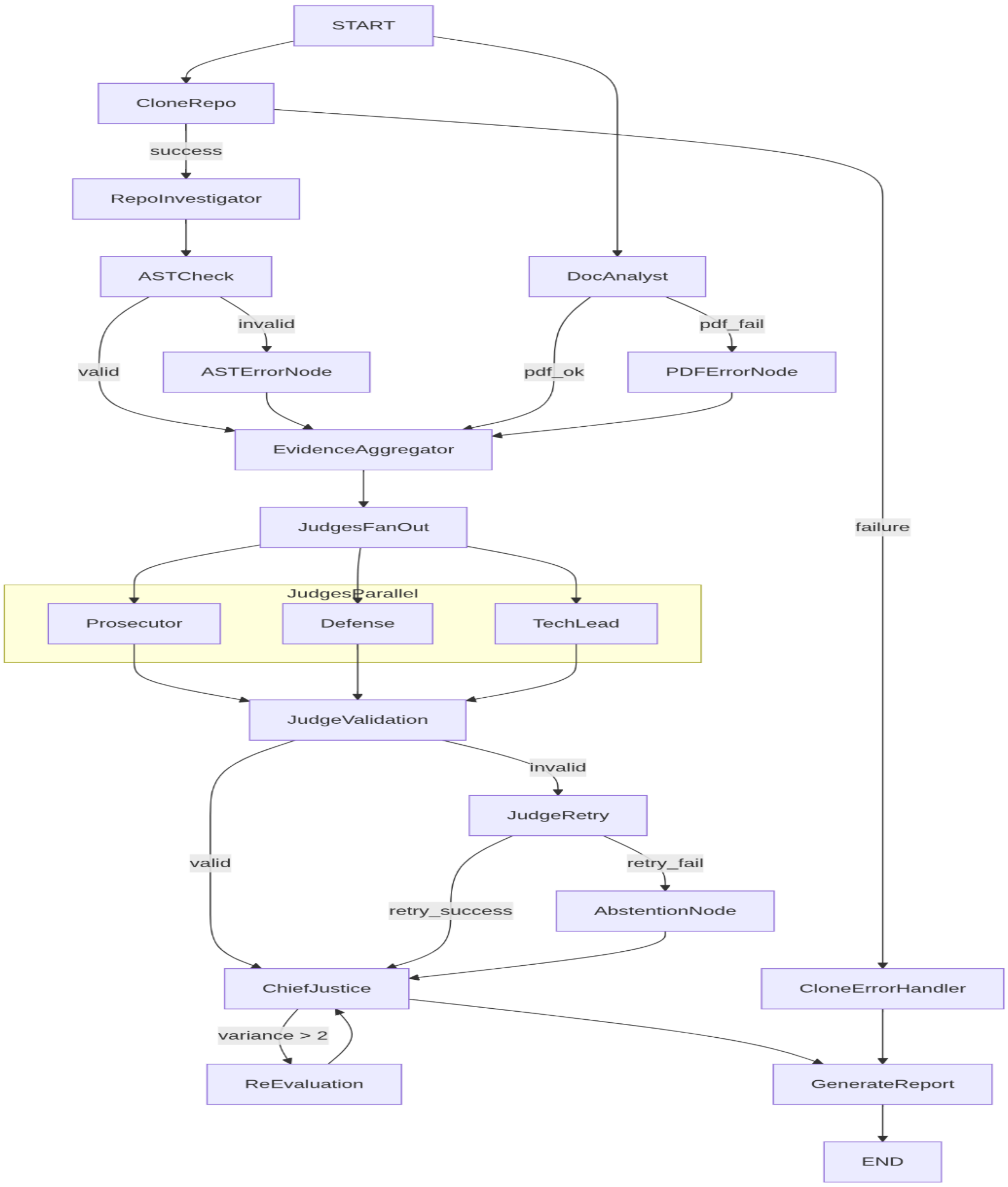
- CriterionResult and AuditReport models exist.
- Rule concepts are documented.
- Deterministic conflict resolution logic is not fully implemented.

Forward Plan

The ChiefJusticeNode will:

1. Aggregate all judicial opinions.
2. Apply deterministic override rules.
3. Produce the final AuditReport.

3 StateGraph Architecture Diagram



StateGraph Architecture Overview

The diagram illustrates a hierarchical LangGraph workflow with two distinct fan-out/fan-in stages and explicit conditional error routing.

In the first stage, the Detectives (RepoInvestigator, DocAnalyst, VisionInspector) execute in parallel after repository cloning. Their outputs converge at the EvidenceAggregator node, ensuring state synchronization before judicial reasoning begins. If cloning or parsing fails, conditional edges route execution to controlled error handlers rather than terminating the system.

In the second stage, the Judges (Prosecutor, Defense, TechLead) analyze the same evidence concurrently. This preserves epistemic independence and enables dialectical tension. All judicial outputs pass through structured validation before reaching the Chief Justice.

The ChiefJusticeNode applies deterministic synthesis rules. If score variance exceeds a defined threshold, the system triggers re-evaluation before rendering a final verdict.

This architecture ensures parallel evidence collection, structured debate, safe failure handling, and rule-based synthesis — rather than a linear grading pipeline

4 Failure Mode & Conditional Routing Strategy

A production-grade auditor must fail safely. The system therefore defines explicit failure modes and conditional routing logic within the StateGraph.

4.1 Repository Cloning Failure

Failure Mode:

- Invalid GitHub URL
- Authentication failure
- Network timeout

Handling Strategy:

- Clone wrapped in tempfile.TemporaryDirectory()
- subprocess.run() with check=True
- If exception occurs → emit Evidence(found=False, rationale="Clone failure")
- Route to CloneErrorHandler node

Graph Behavior: Conditional edge:

- If clone_success == False → skip analysis → generate partial AuditReport with critical failure flag.

4.2 AST Parsing Failure

Failure Mode:

- Syntax errors in repository
- Non-Python repository

Handling Strategy:

- Wrap ast.parse() in try/except
- If parsing fails → record structured Evidence with low confidence
- Continue analysis of remaining files

Graph Behavior: Conditional edge:

- If ast_valid == False → mark "Forensic Limitation" but proceed to Judges.
- System never crashes due to malformed code.

4.3 PDF Parsing Failure

Failure Mode:

- Corrupt PDF
- No text layer

Handling Strategy:

- Fallback to minimal metadata extraction
- Emit Evidence(found=False, rationale="PDF unreadable")

Conditional routing:

- If pdf_ingested == False → Defense cannot claim documentation depth (Rule of Evidence).

4.4 Judge Output Validation Failure

Failure Mode:

- LLM returns free text
- Invalid JSON

- score outside [1–5]

Handling Strategy:

- Judges use .with_structured_output(JudicialOpinion)
- Pydantic validation enforced
- If validation fails → automatic retry (max 2 attempts)
- If retry fails → Judge abstains and opinion marked invalid

Conditional edge:

- If judge_valid == False → route to JudgeRetryNode
- If still invalid → record abstention

This prevents hallucinated judicial reasoning.

4.5 Variance Conflict Trigger

- If score variance > 2:
- Trigger re-evaluation of cited evidence
- Apply deterministic conflict rules
- Generate mandatory dissent summary

This ensures dialectical integrity.

5 Theoretical Foundations

Dialectical Synthesis

The system operationalizes Thesis–Antithesis–Synthesis:

- Prosecutor = Thesis (Critical stance)
- Defense = Antithesis (Effort recognition)
- Chief Justice = Deterministic Synthesis

Synthesis is not LLM averaging. It applies rule-based supremacy:

- Security override
- Fact supremacy
- Variance-triggered dissent

Metacognition

The auditor evaluates not only code but evaluation quality itself:

- Judges critique evidence

- Chief Justice critiques judges
- Variance detection triggers re-analysis

This creates recursive governance — evaluation of evaluation.

Fan-In / Fan-Out as Epistemic Isolation

Parallel branches ensure:

- Independent reasoning
- No cross-contamination of bias
- Aggregation only after evidence completeness

Fan-in ensures state synchronization before synthesis.