B1W2 Interim Report: The Automaton Auditor

Autonomous Code Governance Swarm - LangGraph Implementation

Author: Zerubabel Jember
Repository: https://github.com/Zerubabel-J/autonomous-code-governance-swarm-langgraph
Submission Date: February 25, 2026
Phase Completed: Phase 1 (Linear PoC) + Phase 3 Topology (Parallel Fan-Out Compiled)

Table of Contents

1. System Overview

The Automaton Auditor is a hierarchical multi-agent swarm built on LangGraph that forensically audits AI-generated code repositories.

The system implements a **Digital Courtroom pattern**:

- Detective agents collect structured forensic evidence from a target repository
- Judge agents deliberate using adversarial personas
- A Chief Justice synthesizes a deterministic final verdict and generates a Markdown audit report

The system evaluates five rubric dimensions against a repository's source code:

Evaluation Dimensions

1. Git Forensic Analysis
Measures commit history progression, development cadence, and bulk upload detection.

2. State Management Rigor
Evaluates Pydantic BaseModel schemas, TypedDict usage, and LangGraph reducer correctness.

3. Graph Orchestration Architecture
Assesses StateGraph topology, fan-out/fan-in patterns, and aggregator implementation.

4. Safe Tool Engineering
Validates sandboxed execution, subprocess hygiene, and absence of unsafe patterns such as `os.system()`.

5. Structured Output Enforcement
Checks correct LLM schema binding via `.with_structured_output()`.

2. Architecture Decision Rationale

2.1 LangGraph StateGraph Over Custom Orchestration

LangGraph was selected over a custom orchestration loop because it provides:

- Native parallel fan-out via multi-edge routing
- Typed state containers with reducer semantics
- Graph compilation with fail-fast behavior at build time

A pipeline that crashes during compilation is safer than one that silently drops evidence at runtime.

2.2 Immutable Evidence via Pydantic `frozen=True`

The `Evidence` schema is declared with `ConfigDict(frozen=True)`, making each evidence object immutable once created by the detective node.

Rationale:

Forensic trust must be established at collection time and preserved thereafter. If judge nodes or aggregators could mutate evidence objects, a compromised or hallucinating judge could alter the factual basis of deliberation. Freezing evidence guarantees downstream integrity.

2.3 Deterministic Rules Before Any LLM Call (RISK-1 Mitigation)

The ChiefJustice applies four deterministic Python rules before invoking any LLM:

1. Rule of Security
   Security violations (e.g., `os.system`) cap the score at 3 based solely on AST tool output.
2. Rule of Fact Supremacy
   Opinions citing file locations not confirmed by detective evidence are overruled.
3. Rule of Functionality Weight
   The TechLead score carries 50% weight for the graph_orchestration criterion.
4. Dissent Requirement
   If score variance exceeds 2 points across judges, a dissent summary is mandatory.

Rationale:

LLMs are non-deterministic. Anchoring scoring logic to deterministic rules ensures reproducibility. The LLM is used only for narrative generation and remediation advice — never for correctness-critical decisions.

2.4 AST-Based Forensic Detection Over Regex

All forensic checkers use Python’s `ast` module rather than string matching.

Rationale:

Regex produces false positives when security-relevant strings appear in comments or docstrings. AST walking inspects actual executable structure, ensuring only real function calls trigger detection. This is critical for automated scoring integrity.

2.5 Three Adversarial Judge Personas (RISK-4 Mitigation)

Three judges evaluate identical evidence from structurally different perspectives:

Prosecutor
Mandate: Adversarial gap detection
Philosophy: Trust no one. Assume vibe coding.

Defense
Mandate: Reward intent and effort
Philosophy: Evaluate spirit of implementation.

TechLead
Mandate: Pragmatic artifact evaluation
Philosophy: Judge the artifact, not the struggle.

System prompts share less than 40% vocabulary overlap (validated by automated test) to prevent persona collusion.

Rationale:

Single-judge systems suffer from anchoring bias. A dialectical bench surfaces disagreement and requires explicit dissent handling when divergence exceeds 2 points.

2.6 Parallel-Safe State with Reducers

The AgentState uses LangGraph reducers:

- `operator.ior` for dictionary merges (evidences)
- `operator.add` for list concatenation (opinions)

Rationale:

In parallel fan-out topologies, multiple nodes write simultaneously. Without reducers, state would be overwritten by the last writer. Reducers ensure deterministic accumulation of all outputs.

Evidence keys are namespaced (`repo_investigator_{criterion_id}`), making collisions structurally impossible.

2.7 Rubric Loaded at Runtime from `rubric.json`

No rubric criteria are hardcoded in agent prompts. All rubric content is loaded from `rubric.json`.

Rationale:

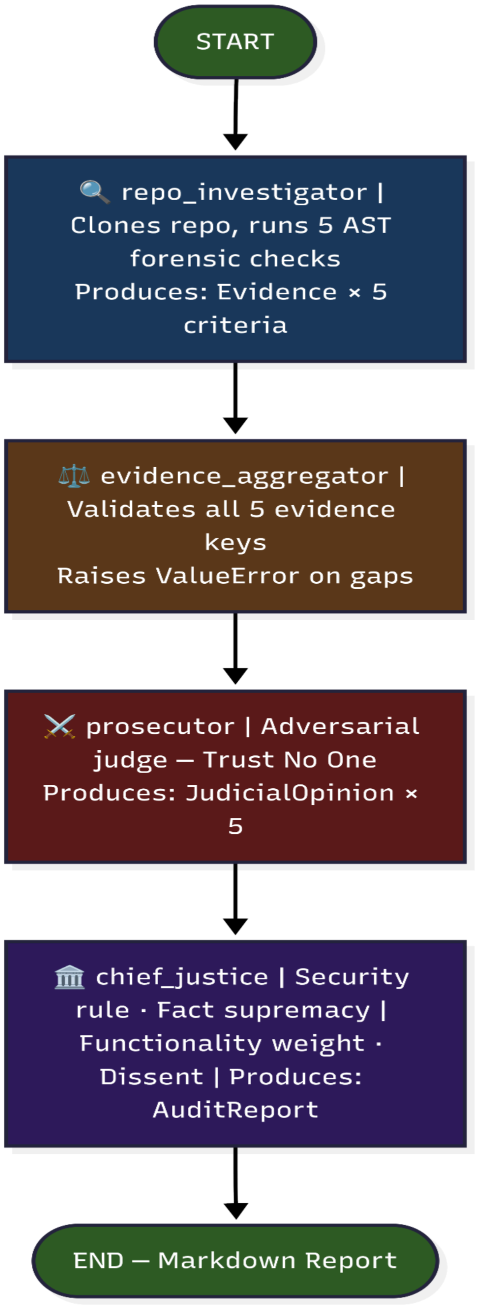
Hardcoded prompts risk divergence when standards evolve. A single canonical rubric file acts as the project’s constitutional source of truth.

3. StateGraph Architecture

3.1 Phase 1 — Linear Topology (Implemented)

Flow:

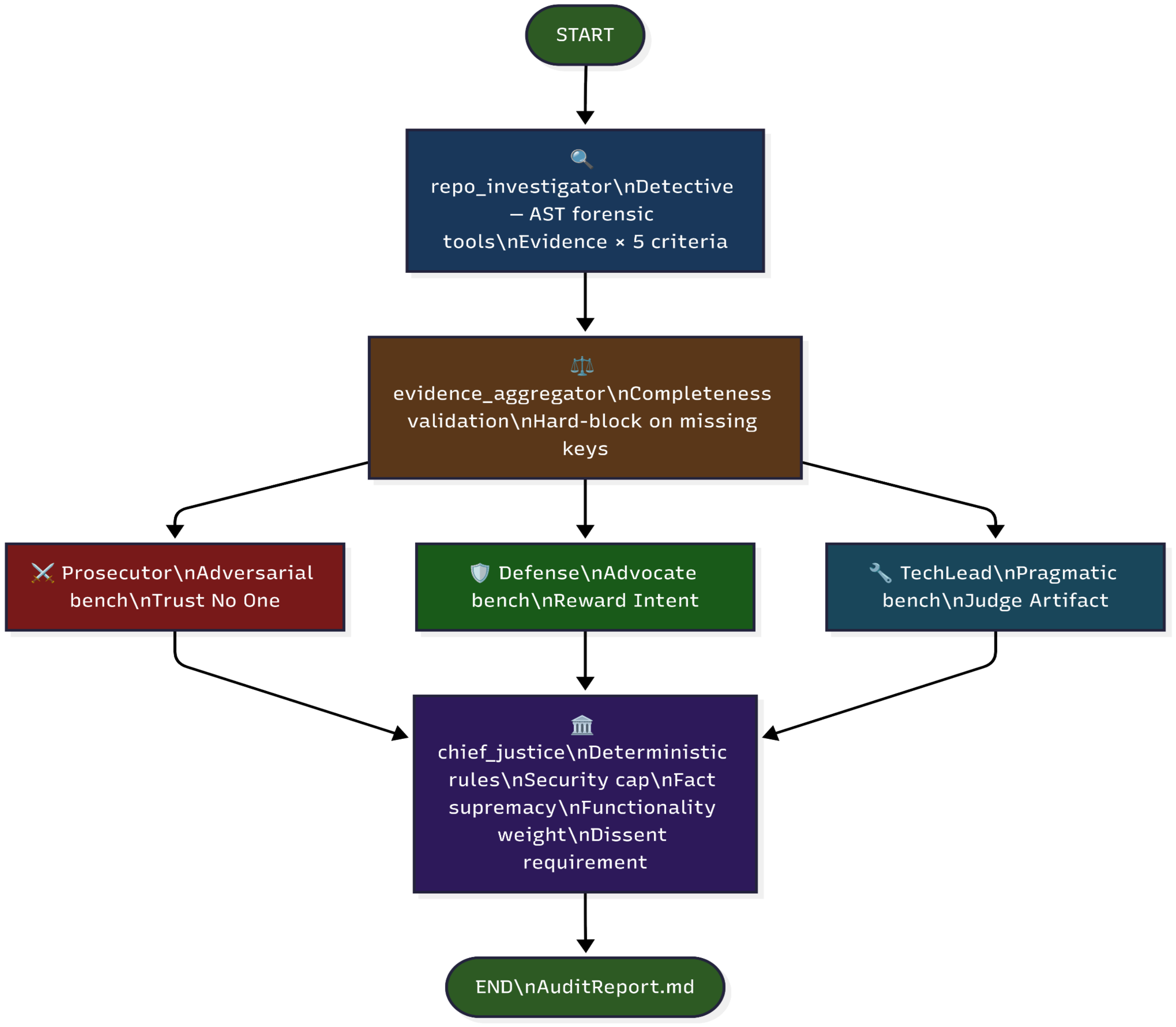
START
→ repo_investigator
→ evidence_aggregator
→ prosecutor
→ chief_justice
→ END (Markdown Report)


START
🔍 repo_investigator | Clones repo, runs 5 AST forensic checks
Produces: Evidence × 5 criteria
⚖️ evidence_aggregator | Validates all 5 evidence keys
Raises ValueError on gaps
⚔️ prosecutor | Adversarial judge — Trust No One
Produces: JudicialOpinion × 5
🏛️ chief_justice | Security rule · Fact supremacy | Functionality weight · Dissent | Produces: AuditReport
END — Markdown Report

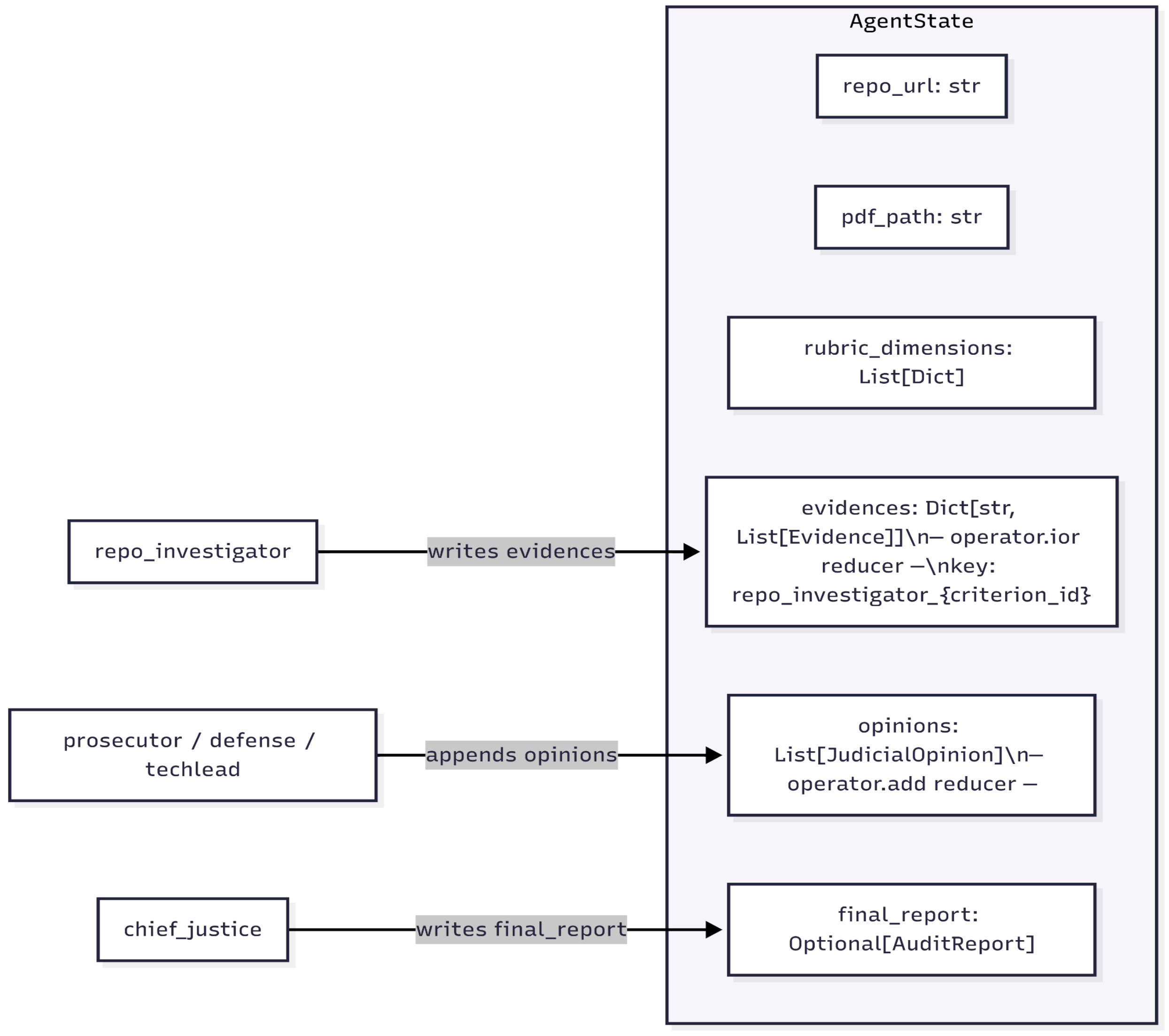
3.2 Phase 3 — Parallel Fan-Out Topology (Compiled)

Flow:

START
→ repo_investigator
→ evidence_aggregator
→ prosecutor
→ defense
→ techlead
→ chief_justice
→ END (AuditReport.md)


START
🔍 repo_investigator\nDetective — AST forensic tools\nEvidence × 5 criteria
⚖️ evidence_aggregator\nCompleteness validation\nHard-block on missing keys
⚔️ Prosecutor\nAdversarial bench\nTrust No One
🛡️ Defense\nAdvocate bench\nReward Intent
🔧 TechLead\nPragmatic bench\nJudge Artifact
🏛️ chief_justice\nDeterministic rules\nSecurity cap\nFact supremacy\nFunctionality weight\nDissent requirement
END\nAuditReport.md

3.3 State Object Flow


AgentState
repo_url: str
pdf_path: str
rubric_dimensions: List[Dict]
evidences: Dict[str, List[Evidence]]\n— operator.ior reducer —\nkey: repo_investigator_{criterion_id}
opinions: List[JudicialOpinion]\n— operator.add reducer —
final_report: Optional[AuditReport]
repo_investigator
writes evidences
prosecutor / defense / techlead
appends opinions
chief_justice
writes final_report

State Fields:

- repo_url: str
- pdf_path: str
- rubric_dimensions: List[Dict]
- evidences: Dict[str, List[Evidence]] (operator.ior reducer)
- opinions: List[JudicialOpinion] (operator.add reducer)
- final_report: Optional[AuditReport]

Write Access:

- repo_investigator → evidences
- prosecutor/defense/techlead → opinions
- chief_justice → final_report

4. Gap Analysis and Forward Plan

4.1 Current Phase Status

Phase 1 — Linear PoC
Status: Complete
Scope: Single detective + Prosecutor + ChiefJustice

Phase 2 — Test Suite
Status: Complete
Scope: 74 tests, all passing, mocked LLM

Phase 3 — Parallel Judges
Status: Compiled, not activated
Scope: Defense + TechLead implemented

Phase 4 — DocAnalyst
Status: Not started

Phase 5 — VisionInspector
Status: Not started

4.2 Gaps Identified by Self-Audit

The system audited itself and produced an overall score of **2.6/5**.

Gap 1 — Git Forensic Analysis (1/5)

Finding: 8 commits pushed within 1 minute flagged as bulk upload.
Root Cause: Development occurred in a single session.
Classification: Process gap, not code defect.
Forward Plan: Future phases developed across distinct sessions to reflect progression.

Gap 2 — Safe Tool Engineering (1/5)

Finding: Prosecutor penalized absence of explicit error-handling metadata.
Root Cause: Forensic tool emits boolean flags but not detailed execution metadata.
Classification: Limitation of single-judge Phase 1.
Forward Plan: Add `has_error_handling`, `has_capture_output`, and `return_code_checked` flags to forensic tool.

Gap 3 — Structured Output Enforcement (1/5)

Finding: Prosecutor hallucinated that `.with_structured_output()` violated requirements.
Root Cause: LLM semantic inversion.
Classification: Structural limitation of single-judge design.
Forward Plan: Activate Phase 3 parallel judges to enable dialectical correction.

4.3 Forward Implementation Plan

Phase 3 — Activate Parallel Judges

- Enable `run_audit(parallel=True)`
- Expected score increase: 2.6 → 3.6–4.0
- File impact: `src/graph.py` (single parameter change)

Forensic Tool Enhancements

Add flags:

- has_error_handling
- has_capture_output
- return_code_checked
- has_retry_logic
- has_fallback

Purpose: Provide richer evidence anchoring for judge scoring.

Phase 4 — DocAnalyst

- Implement PDF text extraction in `src/tools/doc_tools.py`
- Add `doc_analyst_node` in parallel with `repo_investigator`
- Target rubric dimensions: report_accuracy, theoretical_depth

5. Self-Audit Results

Self-audit conducted February 25, 2026.
Report archived at: `audit/report_onself_generated/audit.md`

Results

Git Forensic Analysis — 1/5
Bulk upload pattern detected

State Management Rigor — 5/5
All schemas and reducers confirmed

Graph Orchestration Architecture — 5/5
Parallel fan-out/fan-in confirmed

Safe Tool Engineering — 1/5
Adversarial over-penalization

Structured Output Enforcement — 1/5
LLM hallucination

Overall — 2.6/5
Classification: “Vibe Coder” — reflects Phase 1 single-judge limitation

The 2.6/5 score is expected and explainable. The system correctly identified compliant areas (State Management and Graph Architecture) and exposed limitations inherent to Phase 1 single-judge design rather than implementation defects.

Report generated for:
B1W2 — The Automaton Auditor
GitHub Repository:
https://github.com/Zerubabel-J/autonomous-code-governance-swarm-langgraph