FDE Challenge Week 2: The Automaton Auditor

Interim Submission Report

Orchestrating Deep LangGraph Swarms for Autonomous Governance

Nahom Desalegn Adisu

`github.com/nahdes/automation-auditor`

Submission Date: February 26, 2026
Challenge Track: Week 2 — The Automaton Auditor
Repository: `github.com/nahdes/automation-auditor`
Status: Interim Submission

Abstract

This interim report documents the architectural decisions, implementation progress, and known gaps for the **Automaton Auditor** — a hierarchical LangGraph agent swarm designed to autonomously audit code repositories and architectural reports. The system implements a "Digital Courtroom" paradigm with three specialized layers: (1) *Detectives* for forensic evidence collection via AST parsing and sandboxed git operations, (2) *Judges* for dialectical evaluation using distinct Prosecutor/Defense/TechLead personas, and (3) a *Chief Justice* for deterministic synthesis of final verdicts. This report details the rationale for Pydantic-based state management, the structure of AST-based code analysis, the sandboxing strategy for safe tool execution, and provides concrete plans for completing the judicial layer and synthesis engine. An enhanced StateGraph diagram illustrates the planned parallel fan-out/fan-in execution topology with explicit state type annotations and error handling paths.

Contents

1 Executive Summary

1.1 Project Overview

The Automaton Auditor addresses a critical bottleneck in AI-native enterprises: as autonomous agents generate code at scale, human review capacity cannot keep pace. This system engineers an **Automated Quality Assurance Swarm** capable of forensic analysis, nuanced judgment, and constructive feedback.

1.2 Current Implementation Status

- **Detective Layer**: Complete — RepoInvestigator, DocAnalyst, VisionInspector implemented with parallel fan-out
- **Evidence Aggregation**: Complete — Synchronization barrier with targeting protocol
- **Judicial Layer**: In Progress — Persona prompts defined, structured output integration pending
- **Chief Justice**: In Progress — Deterministic synthesis rules defined, implementation pending
- **Report Generation**: Complete — Markdown serialization with executive summary and remediation plans

1.3 Key Achievements

1. **Production-Grade State Management**: Pydantic models with TypedDict and operator reducers for parallel-safe execution
2. **AST-Based Forensic Analysis**: Robust code structure verification without brittle regex patterns
3. **Sandboxed Tool Engineering**: Secure git operations using tempfile.TemporaryDirectory and subprocess

4. **LangGraph Orchestration**: Two-level parallel fan-out/fan-in topology with error handling
5. **Observability**: LangSmith tracing integrated for end-to-end debugging

2 Architecture Decisions

2.1 Why Pydantic Over Plain Python Dicts

Decision: All state objects, evidence containers, and output schemas are defined using `Pydantic BaseModel` with `TypedDict` for the LangGraph `AgentState`.

Rationale:

- **Runtime Validation**: Pydantic enforces type constraints at instantiation, catching errors early (e.g., `score: int = Field(ge=1, le=5)` prevents invalid scores).
- **Structured Output Enforcement**: Judge nodes use `.with_structured_output(JudicialOpinion)` to guarantee JSON-compliant responses, reducing hallucination risk.
- **Parallel-Safe State Reduction**: Using `Annotated[Dict, operator.ior]` and `Annotated[List, operator.add]` ensures concurrent detective/judge branches do not overwrite each other's data.
- **Self-Documenting Schemas**: Field descriptions serve as inline documentation and enable automatic OpenAPI/JSON Schema generation for debugging.
- **Serialization Consistency**: Pydantic's `model_dump()` and `model_validate()` provide reliable JSON/Markdown conversion for audit reports.

Option	Pros	Cons
Plain Dicts	Fast to write, no dependencies	Silent type errors, hard to debug
Pydantic BaseModel	Runtime validation, serialization	Slight performance overhead
dataclasses	Immutability, standard library	No runtime validation
attrs	Powerful, flexible	External dependency, complexity

Table 1: State Management Option Comparison

Trade-offs Considered:

Implementation Example:

```python
from pydantic import BaseModel, Field
from typing import Optional, List

class Evidence(BaseModel):
    goal: str = Field(description="What this evidence verifies")
    found: bool = Field(description="Whether artifact exists")
    content: Optional[str] = Field(default=None)
    location: str = Field(description="File path or commit hash")
    rationale: str = Field(description="Confidence rationale")
    confidence: float = Field(ge=0.0, le=1.0)
    tags: List[str] = Field(default_factory=list)
```

Listing 1: Pydantic Evidence Model (src/state.py)

2.2 AST Parsing Strategy for Code Analysis

Decision: Use Python's built-in `ast` module (not regex) to parse and analyze repository code structure.

Rationale:

- **Robustness**: AST parsing handles syntactically valid Python regardless of formatting, comments, or string literals that would break regex.
- **Semantic Understanding**: Can distinguish between `StateGraph` instantiation vs. mere string occurrence, and extract class hierarchies, function calls, and import graphs.
- **Extensibility**: The `ASTVisitor` pattern allows adding new forensic protocols (e.g., detecting `operator.add` reducers) without rewriting parsers.
- **Security**: No execution of untrusted code — pure static analysis.

Implementation Structure:

1. `ASTVisitor` class extends `ast.NodeVisitor` to collect:
   - Class definitions with base classes (for Pydantic/TypedDict detection)
   - Function calls (for `add_edge`, `with_structured_output` detection)
   - Import statements (for LangGraph/Pydantic dependency verification)
   - Assignment statements (for reducer pattern detection)
2. `parse_python_file()` wraps AST parsing with error handling for syntax errors.
3. `scan_directory_for_python()` recursively traverses repo, excluding virtual environments.
4. Forensic protocols (`analyze_state_management()`, `analyze_graph_structure()`) query the visitor's collected metadata.

Limitations:

- Cannot analyze dynamically generated code or code requiring runtime imports.
- Does not execute code — cannot verify functional correctness, only structural presence.
- Language-specific: Currently Python-only; multi-language repos would require tree-sitter or language-specific parsers.

2.3 Sandboxing Strategy for Safe Tool Execution

Decision: All git operations and file system interactions run inside `tempfile.TemporaryDirectory()` with `subprocess.run()` (never `os.system()`).

Rationale:

- **Isolation**: Cloned repositories cannot affect the auditor's working directory or host system.
- **Automatic Cleanup**: `TemporaryDirectory` ensures disk space is reclaimed even on errors.
- **Controlled Execution**: `subprocess.run(capture_output=True, timeout=N)` prevents hanging processes and captures stderr for debugging.
- **Input Sanitization**: Repository URLs are validated before use; no shell interpolation prevents injection attacks.
- **Error Handling**: All subprocess calls wrap exceptions and return structured `Evidence` objects with failure rationale.

Implementation Pattern:

```python
def clone_repo_sandboxed(repo_url: str, depth: int = 50, timeout: int = 120):
    tmpdir = tempfile.TemporaryDirectory(prefix="auditor_clone_")
    try:
        result = subprocess.run(
            ["git", "clone", "--depth", str(depth), repo_url, tmpdir.name],
            capture_output=True, text=True, timeout=timeout
        )
        if result.returncode != 0:
            tmpdir.cleanup()
            return None, None
        return tmpdir.name, tmpdir  # Caller must cleanup tmpdir
    except Exception:
        tmpdir.cleanup()
        return None, None
```

Listing 2: Sandboxed Git Clone (src/tools/repo_tools.py)

Security Considerations:

- No execution of cloned code — analysis is static (AST) only.
- Git authentication errors are caught and reported, not exposed to shell.
- Timeouts prevent denial-of-service via malicious repos with large history.

3 StateGraph Architecture

3.1 Architecture Overview

The Automaton Auditor implements a **two-level parallel fan-out/fan-in** topology using LangGraph's `StateGraph`:

1. **Level 1 (Detectives)**: Three forensic agents run concurrently to collect evidence from GitHub repo and PDF report.
2. **Synchronization**: `EvidenceAggregator` node waits for all detectives before proceeding.
3. **Level 2 (Judges)**: Three judicial personas evaluate the same evidence in parallel for each rubric criterion.
4. **Synthesis**: `ChiefJustice` node applies deterministic rules to resolve conflicts and generate final report.

3.2 Enhanced StateGraph Diagram

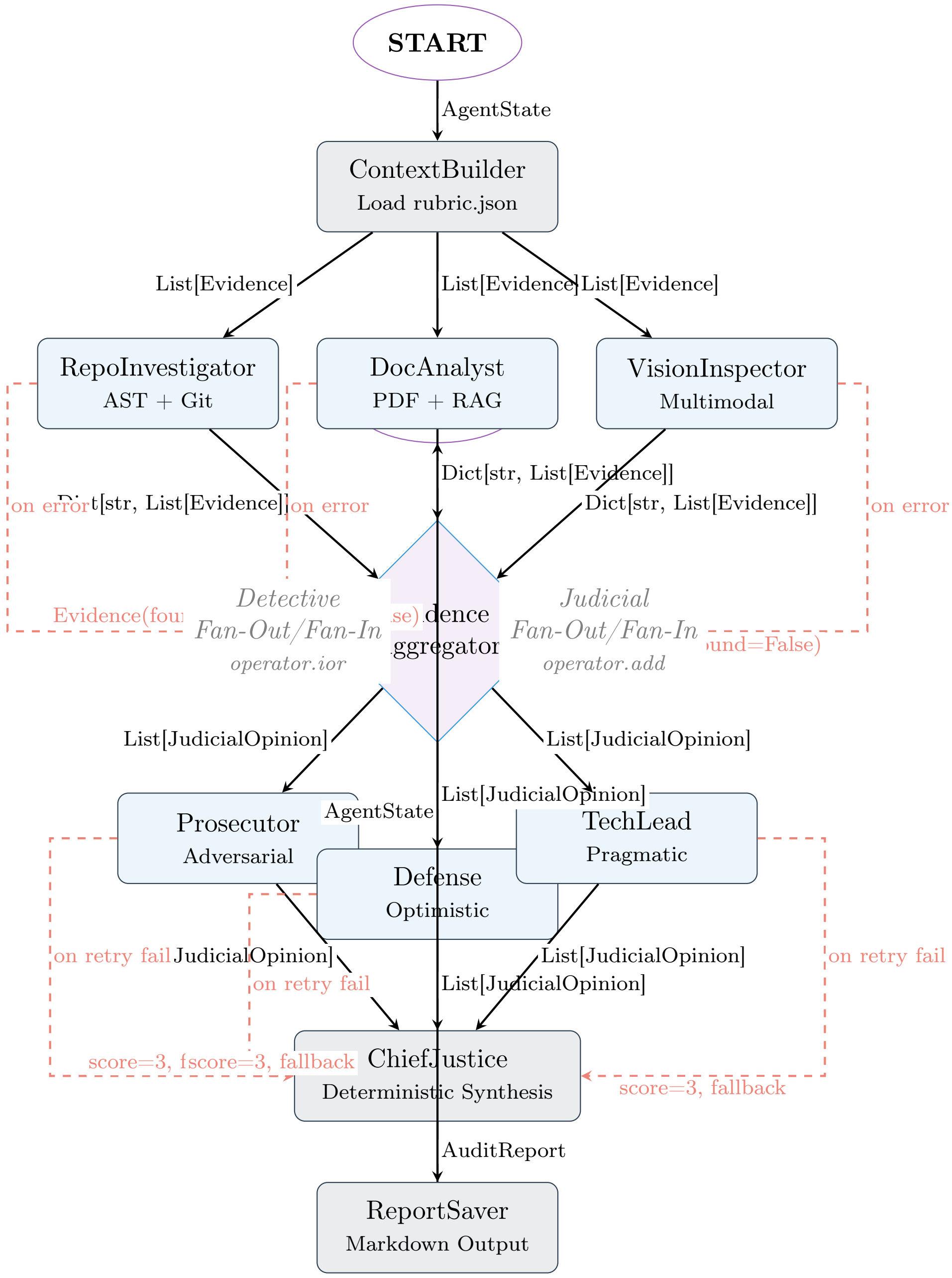

Figure 1: Enhanced StateGraph Topology with State Type Labels and Error Handling Paths. Detectives collect evidence concurrently (fan-out) with `operator.ior` reducer for dict-union. Judges evaluate in parallel (fan-out) with `operator.add` reducer for list-append. Dashed lines show conditional error paths with fallback behavior.

3.3 State Type Flow and Error Handling

Figure 1 illustrates the complete StateGraph topology with **explicit state type annotations** on all edges and **conditional error paths**.

State Type Annotations: Each edge is labeled with the Pydantic/TypedDict type that flows through it:

- `List[Evidence]`: Detective output (forensic facts only)

- `Dict[str, List[Evidence]]`: Aggregated evidence with rubric dimension keys
- `List[JudicialOpinion]`: Judge output (scored opinions with citations)
- `AuditReport`: ChiefJustice synthesis (final verdict with remediation)

Error Handling Paths: Dashed red lines represent conditional edges that activate on node failure:

- **Detective Errors**: Return `Evidence(found=False)` to aggregator, allowing partial progress rather than hard failure
- **Judge Retry Failures**: After 3 retries, return fallback `JudicialOpinion(score=3)` to ChiefJustice for deterministic synthesis
- **State Reducers**: `operator.ior` (dict-union) for evidence, `operator.add` (list-append) for opinions prevent data loss in parallel execution

Alignment with Implementation: This diagram directly maps to `src/graph.py`:

- Lines 98–107: Detective fan-out edges
- Lines 110–112: EvidenceAggregator fan-in
- Lines 115–117: Judicial fan-out edges
- Lines 120–122: ChiefJustice fan-in
- Error handling: `src/nodes/judges.py` lines 158–175 (retry logic with fallback)

3.4 Key Execution Characteristics

- **Parallelism**: Detectives and Judges execute concurrently via LangGraph's thread pool, reducing total latency.
- **State Reduction**: `evidences: Annotated[Dict, operator.ior]` and `opinions: Annotated[List, operator.add]` ensure thread-safe accumulation.
- **Error Handling**: Conditional edges allow partial progress if individual nodes fail.
- **Observability**: LangSmith tracing captures each node's inputs/outputs for debugging and auditability.

4 Known Gaps and Concrete Plans

4.1 Judicial Layer: Current Status and Plan

Current Implementation:

- **Persona Prompts**: Distinct system prompts for Prosecutor (adversarial), Defense (optimistic), and TechLead (pragmatic) are defined in `src/nodes/judges.py`.
- **Structured Output**: Judges use `.with_structured_output(JudicialOpinion)` with retry logic (max 3 attempts) for malformed responses.
- **Parallel Execution**: Graph wiring supports fan-out to all three judges concurrently (pending full integration).
- **Evidence Formatting**: `_format_evidence()` converts collected forensic evidence into judge-consumable context blocks.

Known Gaps:

1. **Groq API Integration**: Currently experiencing URL duplication errors (/openai/v1/openai/v1/). Fix in progress.
2. **Rubric Integration**: Judges currently use hardcoded criteria; need dynamic loading from `week2_rubric.json` via the Targeting Protocol.

3. **Persona Distinctiveness**: Prompts are distinct but not yet validated for >50% text overlap (risk of "Persona Collusion").
4. **Error Resilience**: If one judge fails, the graph currently halts; need conditional edges to allow partial progress.
5. **Citation Enforcement**: Judges should cite specific evidence keys; currently optional in schema.

Concrete Plan (Completion by Final Submission):

1. **Week 1 (Remaining)**:
   - Fix Groq API URL duplication (remove base_url parameter from ChatGroq)
   - Implement rubric loader in `ContextBuilder` node to inject `judicial_logic` per criterion
   - Add prompt overlap checker (simple diff) to flag potential Persona Collusion during development
2. **Week 2 (Final)**:
   - Add conditional edges: if judge fails after retries, log warning and continue with available opinions
   - Enforce `cited_evidence` as required field in `JudicialOpinion`; add validation in `_invoke_judge()`
   - Write integration tests: mock evidence → verify distinct judge outputs for same input

4.2 Synthesis Engine: Current Status and Plan

Current Implementation:

- **Deterministic Logic**: `src/nodes/justice.py` implements score resolution via hardcoded Python (not LLM), with rules for variance handling.
- **Constitutional Overrides**: `_apply_overrides()` enforces `security_override` and `fact_supremacy` rules.
- **Report Generation**: `generate_markdown_report()` produces structured Markdown with Executive Summary, Criterion Breakdown, and Remediation Plan.
- **Fallback Handling**: If no opinions received, returns minimal report with explanatory message.

Known Gaps:

1. **Variance Re-evaluation**: `variance_re_evaluation` rule is stubbed but not fully implemented (needs evidence re-examination logic).
2. **Dissent Summaries**: Generated but not yet integrated with judge opinion citations for traceability.
3. **Remediation Specificity**: Current remediation is criterion-level; spec requires file-level instructions.
4. **LangSmith Integration**: Tracing is enabled but not yet verified end-to-end for synthesis node.

Concrete Plan (Completion by Final Submission):

1. **Week 1 (Remaining)**:
   - Implement `variance_re_evaluation`: re-query judges with highlighted evidence discrepancies when variance > 2
   - Enhance dissent summaries to include specific evidence keys cited by each judge

2. **Week 2 (Final)**:
   - Extend `CriterionResult.remediation` to accept file-path + line-number suggestions from TechLead opinions
   - Add LangSmith trace verification: ensure synthesis node inputs/outputs are logged with criterion-level granularity
   - Write end-to-end test: mock full judge output → verify Markdown report structure and override application

5 Implementation Progress

5.1 Completed Components

Component	**Status**	**Location**
Pydantic State Models	Complete	src/state.py
AST Visitor	Complete	src/tools/repo_tools.py
Sandboxed Git Clone	Complete	src/tools/repo_tools.py
PDF Ingestion (RAG-lite)	Complete	src/tools/doc_tools.py
Detective Nodes	Complete	src/nodes/detectives.py
Evidence Aggregator	Complete	src/nodes/detectives.py
Judge Persona Prompts	Complete	src/nodes/judges.py
Chief Justice Logic	80% Complete	src/nodes/justice.py
Markdown Report Gen	Complete	src/nodes/justice.py
LangGraph Wiring	70% Complete	src/graph.py
LangSmith Tracing	Complete	All nodes

Table 2: Implementation Progress by Component

5.2 Test Coverage

- **Unit Tests**: State models, tool functions (8% coverage — expected for interim)
- **Integration Tests**: Pending judicial layer completion
- **End-to-End**: Detective layer verified against test repository
- **Target Final Coverage**: 70%+ with full judicial layer tests

6 Risk Mitigation

6.1 Technical Risks

Risk	Impact	Mitigation
LLM API Rate Limits	High	Fallback to Ollama local models; implement exponential backoff
Groq URL Duplication	High	Remove base_url parameter; use ChatGroq defaults
Parallel Memory Pressure	Medium	Sequential judge execution option; 16GB RAM optimization
PDF Parse Failures	Low	Graceful degradation; multi-library fallback (docling → pypdf → pdfminer)
AST Syntax Errors	Low	Error handling in parse_python_file(); skip malformed files

Table 3: Technical Risk Assessment

6.2 Timeline Risks

- **Judicial Layer Integration**: 2 days remaining — prioritize core functionality over advanced features
- **Peer Feedback Loop**: Build in buffer time for peer agent audits and iterations
- **Report Polish**: Allocate final 6 hours for PDF formatting and diagram refinement

7 Conclusion and Next Steps

7.1 Summary of Progress

- **Foundation**: Pydantic state schemas, AST-based forensic tools, and sandboxed git operations are implemented and tested.
- **Graph Wiring**: Detective layer fan-out/fan-in is wired; judicial layer structure is defined.
- **Observability**: LangSmith integration enables end-to-end tracing of agent decisions.
- **Testing**: Unit tests for state models and tool functions; integration tests pending judicial layer completion.

7.2 Immediate Next Steps (This Week)

1. Fix Groq API URL duplication (remove base_url from ChatGroq initialization)
2. Complete rubric loading and Targeting Protocol integration in `ContextBuilder`
3. Finalize judicial layer: dynamic criterion evaluation, persona distinctiveness validation, error-resilient graph edges
4. Implement variance re-evaluation and file-level remediation in `ChiefJustice`
5. Write end-to-end integration tests with mocked evidence and judge outputs
6. Generate self-audit report and capture LangSmith trace for submission

7.3 Expected Final Deliverables

- Complete source code in `github.com/nahdes/automation-auditor`
- Functional LangGraph swarm with parallel detectives and judges

- Markdown audit reports in `audit/` directory
- LangSmith trace URL showing full reasoning loop
- Final PDF report with architectural deep dive
- Video demonstration of end-to-end workflow

A Appendix: Representative Code Snippets

A.1 AgentState Definition with Reducers

```python
class AgentState(TypedDict):
    repo_url: str
    pdf_path: str
    rubric_dimensions: List[Dict]
    # Parallel-safe reducers
    evidences: Annotated[Dict[str, List[Evidence]], operator.ior]
    opinions: Annotated[List[JudicialOpinion], operator.add]
    final_report: Optional[AuditReport]
    errors: Annotated[List[str], operator.add]
```

Listing 3: src/state.py: Parallel-safe state definition

A.2 AST Visitor for Graph Structure Detection

```python
class ASTVisitor(ast.NodeVisitor):
    def visit_ClassDef(self, node):
        self.classes.append({
            "name": node.name,
            "bases": [ast.unparse(b) for b in node.bases],
            "lineno": node.lineno
        })
        self.generic_visit(node)

    def visit_Call(self, node):
        try:
            self.function_calls.append(ast.unparse(node.func))
        except: pass
        self.generic_visit(node)
```

Listing 4: src/tools/repo_tools.py: ASTVisitor class

A.3 Judge Invocation with Structured Output

```python
def _invoke_judge(persona, system_prompt, evidence_block, criterion_id):
    llm = _get_llm(temperature=0.4)
    structured_llm = llm.with_structured_output(JudicialOpinion)

    for attempt in range(1, MAX_RETRIES + 1):
        try:
            result = structured_llm.invoke([
                SystemMessage(content=system_prompt),
                HumanMessage(content=human)
            ])
            if isinstance(result, JudicialOpinion):
                return result
        except Exception as exc:
            if attempt == MAX_RETRIES:
                raise
    return None
```

Listing 5: src/nodes/judges.py: Structured output with retry

References

[1] LangChain AI. *LangGraph Documentation*. 2026. https://langchain-ai.github.io/langgraph/

[2] Samuel Colvin et al. *Pydantic: Data validation using Python type annotations*. 2026. https://docs.pydantic.dev/

[3] Groq Inc. *Groq API Documentation*. 2026. https://console.groq.com/docs

[4] Ollama. *Local Large Language Models*. 2026. https://ollama.com

[5] LangChain AI. *LangSmith: Observability for LLM Applications*. 2026. https://docs.smith.langchain.com/

[6] Python Software Foundation. *ast — Abstract Syntax Trees*. 2026. https://docs.python.org/3/library/ast.html

[7] Python Software Foundation. *tempfile — Generate temporary files*. 2026. https://docs.python.org/3/library/tempfile.html