Interim Submission Report (Wednesday 21hr UTC)

Automaton Auditor - Digital Courtroom

Constitutional Multi-Agent Governance System

1. Project Overview

Automaton Auditor is a governance-oriented multi-agent system that audits a target GitHub repository and its accompanying report artifact.
The objective is to replace shallow single-pass grading with a structured, evidence-first workflow that is safer, auditable, and easier to improve over time.

At the interim stage, the implementation focuses on the forensic foundation:

- Parallel detective execution for evidence collection
- Typed state and typed evidence contracts
- Sandboxed repository operations
- Explicit graph-level handling for failure and low-evidence scenarios

Core invariant (interim scope):
START -> Detectives (parallel) -> EvidenceAggregator -> [ErrorCollector | InsufficientEvidence | END]

Why this matters:

- Parallelism improves throughput and evidence diversity
- Typed contracts reduce silent corruption and malformed outputs
- Sandboxing limits risk from untrusted repositories
- Conditional routes prevent silent failures from being mistaken as success

2. Architecture Decisions Made So Far

A. Typed State: Why Pydantic + TypedDict instead of free dicts

Decision:

- Use Pydantic BaseModel contracts (Evidence, JudicialOpinion, AuditReport)
- Use TypedDict AgentState with reducer annotations for merge-safe parallel updates

Rationale:

- Free-form dicts are flexible but error-prone under concurrency
- Missing fields and type mismatches in dicts can propagate unnoticed
- Pydantic introduces runtime validation and clearer schema boundaries

Reducer strategy used in state:

- evidences: operator.ior (map merge)
- opinions: operator.add (list append/merge)
- node_errors: operator.add
- flags: operator.ior

Impact:

- Better determinism in graph execution
- Better debugging and traceability
- Lower risk of branch outputs overwriting each other

Trade-off:

- Slightly more boilerplate
- Meaningfully improved correctness and maintainability

B. AST Parsing Strategy: Why AST over regex

Decision:

- Use AST-based parsing for structural checks in repository forensics

Rationale:

- Regex checks only textual presence and is fragile across formatting/import variations

- AST operates on syntax structure, enabling reliable detection of code patterns

Current AST-based checks:

- StateGraph builder assignment detection
- add_edge and add_conditional_edges call extraction
- fan-out signal detection through edge out-degree
- typed schema structure checks in state.py

Impact:

- Stronger evidence quality for architecture verification
- Fewer false positives than text-only scanning

Trade-off:

- Slight additional parsing complexity
- Better structural reliability for evaluator-grade tooling

C. Sandboxing Strategy for Repository Inspection

Decision:

- Clone target repositories into tempfile.TemporaryDirectory()
- Use subprocess.run with timeout and explicit return-code checks
- Avoid os.system and avoid executing untrusted repository code

Rationale:

- Input repos are untrusted
- Direct cloning/execution in working directory risks mutation and persistence
- Sandboxing gives ephemeral isolation and safer forensic read-only behavior

Additional safeguards currently used:

- GitHub HTTPS URL format validation

- Timeout-bounded subprocess execution
- Auth/repo-not-found error surfacing

Impact:

- Safer and cleaner forensic runtime behavior
- Easier operational debugging when clone/auth fails

Trade-off:

- Fresh clone per run adds overhead
- Safety and reproducibility benefits outweigh runtime cost for audit workflows

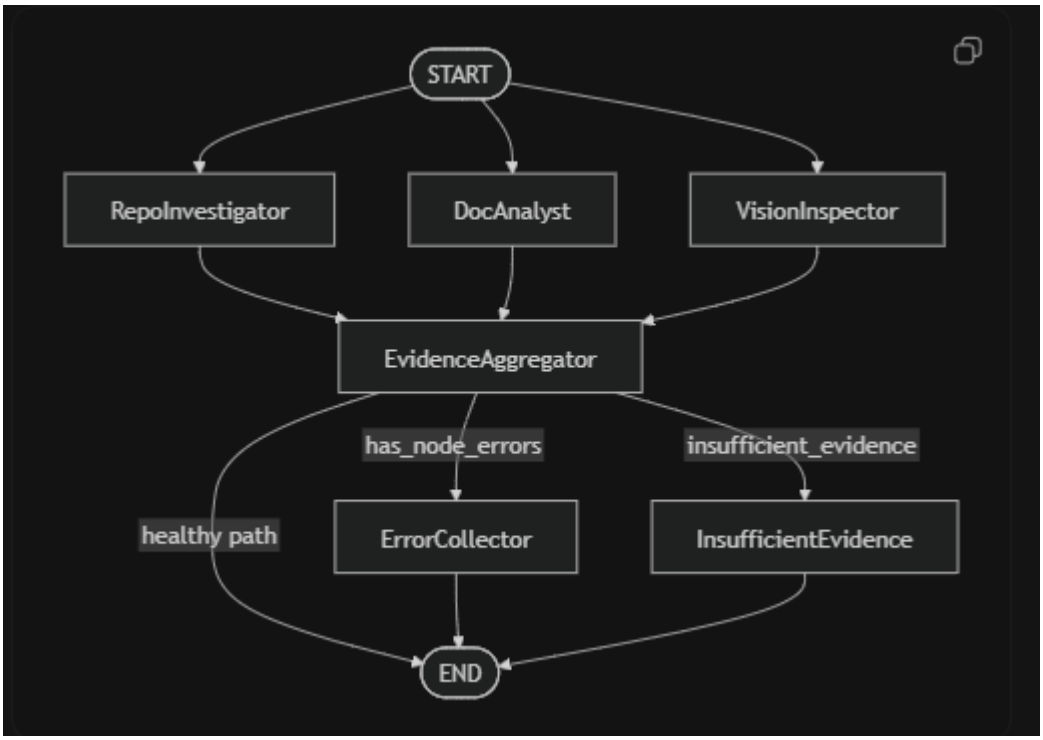
D, Architecture Diagram

The architecture is designed as a governed forensic pipeline rather than a single grading prompt. Execution starts at START and immediately fans out into three detective branches: RepoInvestigator, DocAnalyst, and VisionInspector. This fan-out allows independent evidence collection in parallel from code, document text, and diagram/media context. Each detective produces structured Evidence objects so outputs are typed and merge-safe.

All detective outputs converge at EvidenceAggregator (fan-in), which acts as the control checkpoint. The aggregator summarizes collected evidence, computes basic evidence-quality signals, and inspects error/health flags from upstream nodes. Instead of assuming success, it routes execution conditionally: if node failures were detected, flow goes to ErrorCollector; if evidence volume/quality is insufficient, flow goes to InsufficientEvidence; otherwise it completes through the healthy path to END. This makes non-happy paths explicit and auditable.

State management is intentionally strict. Domain artifacts use Pydantic models (Evidence, JudicialOpinion, AuditReport), while shared graph state uses TypedDict with reducers (operator.ior and operator.add) to prevent parallel branch overwrite issues. Repository analysis is sandboxed through temporary directories and controlled subprocess calls, which reduces risk

from untrusted targets. Overall, this interim architecture establishes the forensic and governance foundation required for the final phase, where judicial fan-out and deterministic Chief Justice synthesis will be layered on top.



3. StateGraph Architecture

A. Interim Implemented Flow

START |---> RepoInvestigator ----\ |---> DocAnalyst -----------> EvidenceAggregator ---> [conditional] ---> END |---> VisionInspector -----/ | +--> ErrorCollector ---> END +--> InsufficientEvidence -> END

B. Data Contracts on Edges

- Detectives -> Aggregator: Dict[str, List[Evidence]]
- Aggregator -> Conditional routes: flags + summarized evidence
- Planned final route: Judges -> ChiefJustice -> AuditReport

C. Why conditional routing is important

The aggregator can route based on explicit flags:

- has_node_errors: route to ErrorCollector
- insufficient_evidence: route to InsufficientEvidence
- otherwise: complete normally

This prevents a common governance failure mode where partial evidence is treated as complete evidence without warning.

4. Known Gaps and Concrete Plan (Judicial Layer + Synthesis)

Current known gaps (interim-accurate)

- judges.py is not implemented yet
- justice.py is not implemented yet
- End-to-end final markdown audit report generation is not complete

Concrete plan: judicial layer

1. Implement Prosecutor, Defense, and TechLead nodes with distinct reasoning objectives.
2. Enforce structured output with JudicialOpinion.
3. Add retry/error handling for malformed structured outputs.
4. Run all judges in parallel on identical criterion evidence.

Concrete plan: synthesis engine

1. Implement deterministic ChiefJustice in Python.
2. Apply explicit conflict rules:
    - Security override
    - Fact supremacy
    - Functionality weighting
    - Dissent requirement

- Variance re-evaluation for highly divergent scores

3. Produce final AuditReport and serialize to markdown.

Expected outcome

This moves the system from forensic collection to full dialectical governance:

- Evidence is collected in parallel
- Interpretations are generated in parallel by opposing judicial personas
- Final verdict is rule-governed and reproducible

5. Risk Register (Interim)

Risk	**Mitigation**
Node exception during evidence collection	Catch exception, emit error evidence, route via ErrorCollector
Insufficient evidence but overconfident interpretation	Evidence count checks + InsufficientEvidence route
Hallucinated report file references	Cross-reference claimed paths against repository file inventory
Structural false positives in code checks	Prefer AST-based structural verification over regex-only checks

6. Conclusion

The interim implementation establishes a robust forensic base: typed state, parallel detective orchestration, sandboxed repository inspection, and explicit failure governance.
The remaining work is focused and clearly scoped: add judicial personas, deterministic synthesis

rules, and final report rendering. This progression keeps the architecture aligned with rubric expectations while avoiding overclaims at the interim stage.