FDE Challenge Week 2: The Automaton Auditor

Author: Mama Mohammed
Date: February 2026
Repo: https://github.com/MamaMoh/TRP1-Challenge-Week-2

1. Introduction

This report documents the design and implementation of Automaton Auditor, developed for Week 2 of the FDE (Forensic Digital Evaluation) Challenge.

The objective of Week 2 is to transition from managing a single "Silicon Worker" (Week 1) to governance at scale. Instead of building agents that produce artifacts, the challenge focuses on building a system that evaluates artifacts — specifically GitHub repositories and accompanying PDF reports — against a structured rubric.

To meet this objective, the system is designed as a multi-agent evaluation pipeline based on a "Digital Courtroom" architecture. It uses a layered LangGraph StateGraph to orchestrate:

- Forensic evidence gathering by parallel detective agents
- Adversarial judicial reasoning by three distinct judge personas
- Deterministic synthesis by a Chief Justice node into a final audit report

This report covers: challenge context and objectives; system design and architecture; the StateGraph orchestration model (with full diagrams and wiring); project structure; evaluation-criteria alignment; and gap analysis with a forward plan. All explanations and diagrams are included in this document so that the design and behaviour can be understood without opening other files.

2. The Challenge Context

In an AI-native enterprise, autonomous agents can generate code at a scale that exceeds human review capacity. The bottleneck shifts from **production** to **evaluation**.

The challenge is therefore to build an **Automated Auditor Swarm** capable of:

1. Forensic analysis
Verifying that claimed artifacts exist and meet structural expectations (e.g. typed state, graph orchestration, safe tool usage). Evidence is collected via tools (git clone, AST parsing, PDF parsing) and recorded as structured objects, not freeform text.

2. Nuanced judgment
Applying a complex rubric across multiple dimensions using distinct perspectives. A single "grader" prompt is insufficient; the system uses separate Prosecutor, Defense, and Tech Lead personas so the same evidence is interpreted in adversarial, charitable, and pragmatic ways.

3. Constructive feedback
Producing actionable remediation guidance (file-level, criterion-level) rather than a simple pass/fail. The output includes per-criterion scores, judge opinions, dissent summaries when variance is high, and a consolidated remediation plan.

Inputs

1. A **GitHub repository URL** (the codebase under audit)
2. A **PDF report** (e.g. architectural description or self-evaluation) — provided as a local path or URL (Google Drive links are supported and converted to direct download)

Output

A structured audit report (Markdown and JSON) including:

- An executive summary and overall score
- Per-criterion scores with judge opinions (Prosecutor, Defense, Tech Lead) and cited evidence
- Dissent summaries where score variance between judges exceeds a threshold
- A remediation plan (consolidated and per criterion)

Conceptual framing: Digital Courtroom

- **Detectives** gather evidence and do not opine; they output structured `Evidence` objects keyed by rubric dimension.
- **Judges** interpret evidence through distinct personas and output structured `JudicialOpinion` objects (score, argument, cited evidence).

- A **Chief Justice** synthesizes the final verdict using **deterministic rules** (security override, fact supremacy, variance/dissent handling), not another LLM call, so the outcome is auditable and repeatable.

3. Project Overview

Automaton Auditor is implemented as a **LangGraph-based** multi-agent orchestration system. It:

- Accepts a repository URL and a PDF (local path or URL; Google Drive links supported)
- Loads a machine-readable rubric (default: `rubric/week2_rubric.json`) that defines dimensions, forensic instructions, success/failure patterns, judicial logic per dimension, and synthesis rules
- Executes a **three-layer** evaluation pipeline (detectives → aggregation and conditional routing → judges → Chief Justice)
- Produces structured audit reports (Markdown and JSON) in a designated directory (e.g. `audit/report_onself_generated/` or `audit/report_onpeer_generated/`)
- Optionally compares the current run's report with a peer report when `--compare <path>` is provided

3.1 Three-Layer Evaluation Pipeline

Layer 1 — Detective layer (parallel forensic agents)

Three detective nodes run **in parallel** from a single entry node. Each has a distinct forensic role and writes into shared state using **reducers** so that evidence from all three is merged without overwriting.

- **RepoInvestigator**
  Clones the repository into a sandboxed temporary directory (using `tempfile.TemporaryDirectory()` and `subprocess.run` for `git clone`). It then runs AST-based verification on the codebase: state models (Pydantic/TypedDict, reducers), graph structure (StateGraph, fan-out/fan-in, conditional edges), safe tool engineering (no `os.system` calls; sandboxing and subprocess usage), and structured output (judges using `with_structured_output(JudicialOpinion)`). It also runs git history analysis (commit count, progression). Evidence is keyed by rubric dimension ID (e.g. `git_forensic_analysis`, `state_management_rigor`, `graph_orchestration`) and appended to the shared `evidences` dictionary via the `operator.ior` reducer.

- **DocAnalyst**
  Parses the PDF (via pypdf or docling), extracts text, and runs keyword search for rubric-relevant terms (e.g. "Dialectical Synthesis", "Fan-In", "Metacognition"). It also **cross-references file paths** mentioned in the PDF against the actual repository: the system obtains a full list of repo files (by cloning the repo in this node, since detectives run in parallel) and checks each path extracted from the PDF. The result is

evidence for “Report Accuracy” (verified vs hallucinated paths) and “Theoretical Depth” (keyword presence and context). Evidence is merged into the same shared `evidences` state.

- **VisionInspector**
  Checks whether the repository contains an **architecture diagram** (e.g. Mermaid flowchart in `docs/architecture.md` or in the README). It clones the repo, scans those files for patterns such as `flowchart`, `graph`, or `stateDiagram`, and returns evidence for the “Architectural Diagram Analysis” dimension. This allows the rubric to be satisfied when the diagram lives in the repo rather than only in the PDF.

All detectives output **structured `Evidence` objects** (goal, found, content, location, rationale, confidence). The graph state uses `Annotated[Dict[str, List[Evidence]], operator.ior]` so that parallel detective outputs are merged by dictionary union.

Layer 2 — Judicial layer (parallel persona-based reasoning)

Three judge nodes run **in parallel** after evidence aggregation. Each receives the **same** merged state (all evidence and rubric dimensions) and scores **every** rubric dimension from a fixed persona.

- **Prosecutor**
  Persona: critical, “trust no one,” assume vibe coding. Prompting emphasizes looking for gaps, security flaws, and rubric violations. Tends to argue for lower scores when evidence is incomplete or when structural requirements (e.g. parallel execution, Pydantic) are not clearly met.

- **Defense**
  Persona: charitable, reward effort and intent. Prompting emphasizes creative workarounds, deep thought, and the “spirit of the law.” Tends to argue for higher scores when the architecture shows intent even if implementation is imperfect.

- **Tech Lead**
  Persona: pragmatic, focus on maintainability and production readiness. Acts as tie-breaker; evaluates technical debt, reducer usage, and whether the system would scale. Provides remediation-oriented reasoning.

Each judge invokes the LLM with **`with_structured_output(JudicialOpinion)`** so that the model returns a Pydantic-validated object (judge name, criterion_id, score 1–5, argument, cited_evidence list). The graph state uses `Annotated[List[Dict], operator.add]` for opinions so that all three judges’ outputs are concatenated. (Opinions are stored as dicts, i.e. `model_dump()`, to avoid Pydantic serialization issues when LangGraph merges state; the Chief Justice converts them back to `JudicialOpinion` instances.)

Layer 3 — Chief Justice (deterministic synthesis)

The Chief Justice node does **not** call an LLM. It:

- Reads all opinions from state and groups them by rubric dimension.
- For each dimension, applies **hardcoded rules**:
  - **Security override**: If the Prosecutor (or evidence) indicates a confirmed security issue (e.g. unsanitized `os.system`), the score for that dimension is capped (e.g. at 3) regardless of Defense or Tech Lead.
  - **Fact supremacy**: If the Defense argues for a claim that the evidence does not support (e.g. "deep metacognition" but no supporting evidence), the Defense is overruled; evidence wins over interpretation.
  - **Functionality weight**: For architecture-related dimensions, the Tech Lead's assessment of modularity and correctness carries high weight.
  - **Variance and dissent**: When the score variance among the three judges exceeds 2 (e.g. Prosecutor 1, Defense 5), the Chief Justice records a dissent summary explaining the conflict and uses a tie-breaker rule (e.g. Tech Lead's score or "fact supremacy") to set the final score.
- Builds a **`CriterionResult`** per dimension (dimension_id, dimension_name, final_score, judge_opinions, dissent_summary, remediation).
- Computes the overall score (e.g. mean of final scores), builds an executive summary string, and assembles a consolidated remediation plan for criteria below threshold.
- Constructs the final **`AuditReport`** (repo_url, executive_summary, overall_score, criteria, remediation_plan) and returns it in state.

The main program then serializes this `AuditReport` to Markdown (and JSON) and writes it to the chosen output directory.

4. System Execution Flow

Step 1 — CLI invocation

The user runs the application (e.g. `python main.py` or `uv run python main.py`) with:

- **`--repo`** (required): GitHub repository URL to audit
- **`--pdf`** (required): Path to the PDF report or a URL (e.g. Google Drive link)
- **`--output`** (optional): Directory for the audit report; default is `audit/report_onself_generated/`
- **`--rubric`** (optional): Path to rubric JSON; default is `rubric/week2_rubric.json`
- **`--compare`** (optional): Path to a peer audit report; after the run, a short comparison (score diff, criterion diffs, peer issues) is printed
- **`--verbose`**, **`--trace`**: Logging and LangSmith tracing

Step 2 — Setup

- The PDF input is resolved: if it is a URL, it is downloaded (Google Drive share links are converted to direct-download URLs) and the local path is stored; the original URL or path is kept for display in the report (`pdf_display`).
- The rubric is loaded from disk; it contains a list of dimensions (each with id, name, target_artifact, forensic_instruction, success_pattern, failure_pattern, judicial_logic) and synthesis_rules (security_override, fact_supremacy, dissent_requirement, etc.).
- Environment configuration is loaded (e.g. `OPENAI_API_KEY` or `OPENROUTER_API_KEY` from `.env` or `.env.example`).
- The StateGraph is built by calling `build_auditor_graph()`: all nodes and edges are registered, including the conditional edges after the evidence aggregator.
- Initial state is constructed: repo_url, pdf_path, pdf_display, rubric_path, rubric_dimensions, synthesis_rules, empty evidences, opinions, errors, and final_report None.

Step 3 — Parallel detectives

- The graph's entry point is the `start` node; from there three edges fan out to `repo_investigator`, `doc_analyst`, and `vision_inspector`. LangGraph runs these three nodes in parallel.
- RepoInvestigator clones the repo into a temp dir, runs `analyze_git_history`, `verify_state_models`, `analyze_graph_structure`, `verify_safe_tool_engineering`, `verify_structured_output`, and optionally scans `judges.py` and `justice.py` for judicial nuance and Chief Justice logic. It returns a partial state update `{"evidences": {...}, "errors": [...]}` if any. The reducer `operator.ior` merges the evidences dict; `operator.add` appends any errors.
- DocAnalyst parses the PDF, gets a repo file list (by cloning the repo again, since it runs in parallel), runs `extract_keywords` and `verify_file_claims`, and returns evidences for theoretical_depth and report_accuracy dimensions.
- VisionInspector clones the repo and checks for Mermaid in `docs/architecture.md` or README, returning evidence for the swarm_visual / architectural diagram dimension.
- After all three complete, the state holds a merged `evidences` dictionary and any `errors` list.

Step 4 — Evidence aggregation and conditional routing

- The **evidence_aggregator** node is the fan-in: all three detectives have edges into it. The node does not add new data; it is the synchronization point. State is already merged.
- A **conditional edge** leaves the evidence_aggregator. The function `route_after_aggregator(state)` inspects `state["errors"]` and `state["evidences"]`. If there are any errors, or if evidences is empty, the next node is **handle_failure_or_missing**; otherwise it is **to_judges**.

- **handle_failure_or_missing** appends a single degradation message to `state["errors"]` (e.g. "Audit degraded: evidence missing or node failure during

detective phase; proceeding with partial evaluation") and returns. From here, the graph still fans out to the three judges so that a report can be generated that explicitly reflects the failure. **to_judges** is a passthrough node that returns state unchanged. Both nodes have edges to all three judge nodes.

Step 5 — Parallel judges

- Prosecutor, Defense, and Tech Lead each run in parallel. Each receives the full state (all evidences, all rubric dimensions). For each rubric dimension, each judge builds a prompt that includes the dimension name, the forensic instruction or judicial logic from the rubric, and the relevant evidence text (with braces escaped so that LangChain's prompt template does not interpret them as placeholders). The LLM is called with `with_structured_output(JudicialOpinion)`. The returned opinion is appended to state as a dict (`opinion.model_dump()`). The reducer `operator.add` concatenates the three judges' opinion lists.

Step 6 — Chief Justice

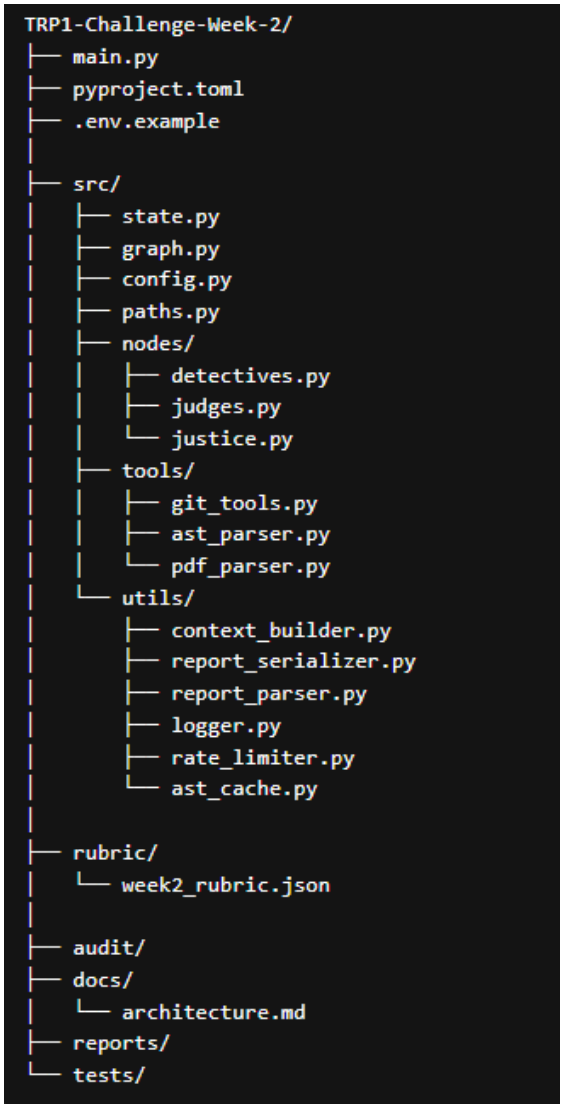
1. The Chief Justice node reads `state["opinions"]`, converts each item to a `JudicialOpinion` instance if it is a dict, and groups opinions by criterion_id. For each criterion it has three opinions (Prosecutor, Defense, Tech Lead). It applies the synthesis rules: security override (cap score if security flaw), fact supremacy (overrule Defense when evidence contradicts), and variance handling (record dissent when variance > 2, use Tech Lead or rule-based tie-breaker). It builds `CriterionResult` objects (final_score, judge_opinions, dissent_summary, remediation) and then the single `AuditReport` (executive_summary, overall_score, criteria, remediation_plan). The return value is `{"final_report": audit_report}`.

Step 7 — Output

- The main program reads `final_state["final_report"]`, serializes it to Markdown via `serialize_report_to_markdown(audit_report)` (which produces the metadata table, score overview, criterion breakdown with judge opinions and dissent, and remediation plan), and writes the result to `<output_dir>/audit_report.md`. It also writes `audit_report.model_dump()` as JSON to `<output_dir>/audit_report.json`. If `--compare` was set, it calls `compare_reports(current_report_path, peer_report_path)` and prints overall score difference, criterion-by-criterion differences, and a short list of issues from the peer report.

5. Project Structure

The repository is organized as follows.



6. Architecture and StateGraph Design

The system is implemented using a LangGraph StateGraph so that: parallelism is explicit (detective and judge nodes fan out from common sources); state merging is explicit (evidences `operator.ior`, opinions and errors `operator.add`); execution order is deterministic (entry → detectives → evidence_aggregator → conditional → judges → chief_justice → END); conditional routing allows handling "evidence missing" or "node failure" without aborting.

6.1 Design Decisions

- **Typed state (TypedDict + Pydantic):** AgentState is TypedDict; Evidence, JudicialOpinion, CriterionResult, AuditReport are Pydantic. Reducers required for parallel merge.

- **Three detectives:** RepoInvestigator (code/repo), DocAnalyst (PDF/cross-reference), VisionInspector (diagram); clear forensic roles and parallel execution.
- **Three judges + Chief Justice:** Distinct personas (adversarial, charitable, pragmatic); Chief Justice is pure Python rules, not LLM.
- **Conditional edges:** `route_after_aggregator` → `to_judges` or `handle_failure_or_missing`; degradation path still leads to judges so the report can state audit degradation.
- **Sandboxing for repo clone:** All git clones use a temp directory (never cwd), `subprocess.run` with list args (no shell), URL validation, timeout, and post-clone git check (see Section 7.1 for full rationale, alternatives, and trade-offs).

Section 7.1 elaborates on why these choices were made over alternatives and the trade-offs involved.

6.2 High-Level Flow (text diagram)

```
start → [RepoInvestigator | DocAnalyst | VisionInspector] (parallel)
      → evidence_aggregator (fan-in)
      → route_after_aggregator → to_judges OR handle_failure_or_missing
      → [Prosecutor | Defense | Tech Lead] (parallel)
      → chief_justice → END
```

6.3 Fan-Out / Fan-In and Conditional Edges

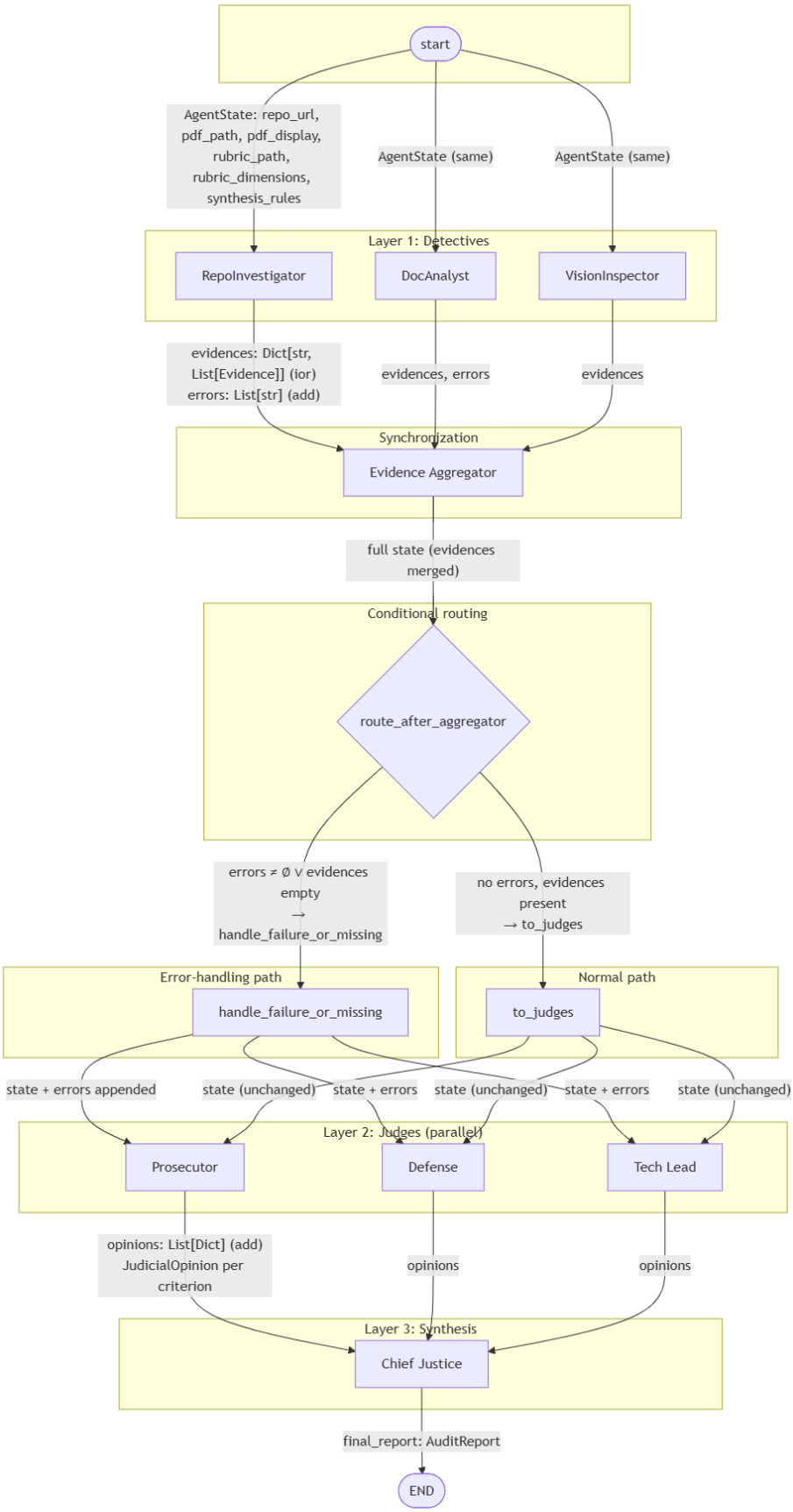
- **Fan-out:** From start → three Detectives; from post-aggregator path → three Judges.
- **Fan-in:** All Detectives → Evidence Aggregator; all Judges → Chief Justice.
- **Conditional:** After Evidence Aggregator, `route_after_aggregator(state)` chooses `to_judges` (no errors, evidences present) or `handle_failure_or_missing` (errors or empty evidences). Both branches then fan out to the same three judge nodes.

6.4 StateGraph Diagrams (with state types on edges and conditional/error paths)

The following diagrams visually represent **state types or data labels on the edges** and **explicit conditional edges and error-handling paths** as required by the rubric.

Diagram 1: Full flow with state types on every edge

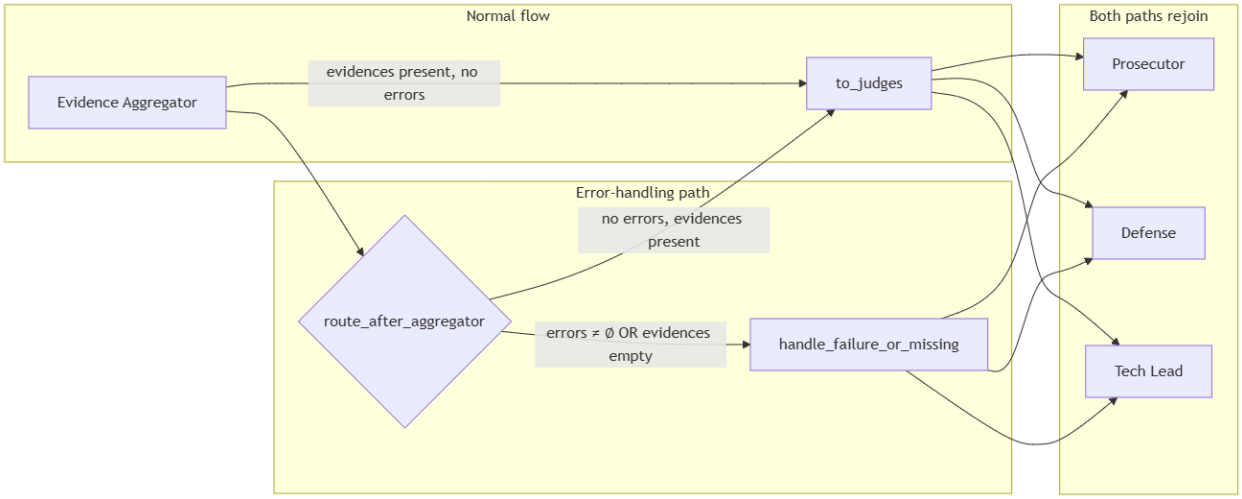
Every edge is labeled with the **AgentState** keys (and types where helpful) that flow along it. The conditional node and the two outgoing branches (normal vs error path) are explicitly shown.


start
AgentState: repo_url, pdf_path, pdf_display, rubric_path, rubric_dimensions, synthesis_rules
AgentState (same)
AgentState (same)
Layer 1: Detectives
RepoInvestigator
DocAnalyst
VisionInspector
evidences: Dict[str, List[Evidence]] (ior)
errors: List[str] (add)
evidences, errors
evidences
Synchronization
Evidence Aggregator
full state (evidences merged)
Conditional routing
route_after_aggregator
errors ≠ ∅ ∨ evidences empty → handle_failure_or_missing
no errors, evidences present → to_judges
Error-handling path
handle_failure_or_missing
Normal path
to_judges
state + errors appended
state (unchanged)
state + errors
state (unchanged)
state + errors
state (unchanged)
Layer 2: Judges (parallel)
Prosecutor
Defense
Tech Lead
opinions: List[Dict] (add) JudicialOpinion per criterion
opinions
opinions
Layer 3: Synthesis
Chief Justice
final_report: AuditReport
END

- **Edge labels:** Each edge shows the AgentState keys (and reducer/type where relevant) that flow on that edge. Reducers: `evidences` use `operator.ior` (dict merge); `opinions` and `errors` use `operator.add` (list append).
- **Conditional node:** `route_after_aggregator` has two outgoing edges: one to **handle_failure_or_missing** (error path) when `errors` non-empty or `evidences` empty; one to **to_judges** (normal path) otherwise.
- **Error-handling path:** The branch into `handle_failure_or_missing` is the explicit error path; that node only appends a degradation message to `errors` and does not change evidence; both paths then feed the same three judge nodes.

Diagram 2: Conditional and error-handling path highlighted

This diagram emphasizes the **conditional edges** and the **error-handling path** so that they are visually distinct from the normal flow.



- **Conditional edges:** The diamond `route_after_aggregator` has two outcomes: (1) **Error path:** when there are errors or no evidence → `handle_failure_or_missing`, which appends a degradation message to `state.errors`. (2) **Normal path:** when evidence exists and no errors → `to_judges` (passthrough). Both paths then fan out to Prosecutor, Defense, and Tech Lead so that the judicial layer and Chief Justice always run, and the report can record a degraded audit when applicable.

Diagram 3: State types summary (AgentState schema on edges)

Edge / segment	State keys and types flowing
start → Detectives	`repo_url`, `pdf_path`, `pdf_display`, `rubric_path`, `rubric_dimensions`, `synthesis_rules` (inputs; no reducers)
Detectives → Evidence Aggregator	`evidences: Dict[str, List[Evidence]]` (reducer: `operator.ior`), `errors: List[str]` (reducer: `operator.add`)
Evidence Aggregator → conditional	Full merged state (evidences, errors, plus all inputs)

Edge / segment	State keys and types flowing
Conditional → to_judges	Full state (unchanged)
Conditional → handle_failure_or_missing	Full state; node returns `{"errors": state.errors + [degradation_msg]}`
to_judges / handle_failure_or_missing → Judges	Full state (including any appended errors)
Judges → Chief Justice	`opinions: List[Dict]` (reducer: `operator.add`; each dict is JudicialOpinion.model_dump())
Chief Justice → END	`final_report: AuditReport`

These three elements (Diagram 1, Diagram 2, and the state-types table) together provide: (1) state types or data labels on the edges, and (2) explicit representation of conditional edges and error-handling paths.

6.5 Graph Wiring (nodes and edges as in code)

```
workflow = StateGraph(AgentState)
workflow.add_node("start", start_node)
workflow.add_node("repo_investigator", repo_investigator_node)
workflow.add_node("doc_analyst", doc_analyst_node)
workflow.add_node("vision_inspector", vision_inspector_node)
workflow.add_node("evidence_aggregator", evidence_aggregator_node)
workflow.add_node("to_judges", to_judges_node)
workflow.add_node("handle_failure_or_missing",
handle_failure_or_missing_node)
workflow.add_node("prosecutor", prosecutor_node)
workflow.add_node("defense", defense_node)
workflow.add_node("tech_lead", tech_lead_node)
workflow.add_node("chief_justice", chief_justice_node)

workflow.set_entry_point("start")
workflow.add_edge("start", "repo_investigator")
workflow.add_edge("start", "doc_analyst")
workflow.add_edge("start", "vision_inspector")
workflow.add_edge("repo_investigator", "evidence_aggregator")
workflow.add_edge("doc_analyst", "evidence_aggregator")
workflow.add_edge("vision_inspector", "evidence_aggregator")
workflow.add_conditional_edges("evidence_aggregator", route_after_aggregator,
    path_map={"to_judges": "to_judges", "handle_failure_or_missing":
"handle_failure_or_missing"})
workflow.add_edge("to_judges", "prosecutor")
workflow.add_edge("to_judges", "defense")
workflow.add_edge("to_judges", "tech_lead")
workflow.add_edge("handle_failure_or_missing", "prosecutor")
workflow.add_edge("handle_failure_or_missing", "defense")
```

```
workflow.add_edge("handle_failure_or_missing", "tech_lead")
workflow.add_edge("prosecutor", "chief_justice")
workflow.add_edge("defense", "chief_justice")
workflow.add_edge("tech_lead", "chief_justice")
workflow.add_edge("chief_justice", END)
```

7. Evaluation Criteria Alignment

7.1 Architecture Decision Rationale

The following explains why specific choices were made, what alternatives were considered, and what trade-offs were accepted.

Hierarchical StateGraph

- **Choice:** A LangGraph `StateGraph` with explicit nodes (start, detectives, evidence_aggregator, conditional, judges, chief_justice) and edges defining fan-out, fan-in, and conditional routing.
- **Alternatives considered:** (1) A linear script that runs detectives sequentially then judges sequentially — simpler to read but no explicit parallelism and no clear place to attach conditional logic. (2) A single "super-agent" that uses tools for git, PDF, and diagram checks — fewer moving parts but all evidence and control flow in one prompt; state merging and error handling become implicit and harder to audit. (3) A DAG in another framework (e.g. Prefect, Airflow) — would give parallelism and conditioning but would require a different state model and would not align with the rubric's LangGraph expectation.
- **Trade-offs:** The graph is more complex than a linear script and requires understanding of reducers and conditional edges. In return, parallelism is explicit (runtime can execute detectives and judges in parallel), state flow is visible in the diagram, and conditional routing (e.g. evidence missing) is a first-class construct. We accepted the added complexity to satisfy the rubric's requirement for explicit fan-out/fan-in and conditional edges and to keep the pipeline auditable.

Typed state (TypedDict + Pydantic)

- **Choice:** Top-level graph state is a `TypedDict` (`AgentState`); nested values use Pydantic models (`Evidence`, `JudicialOpinion`, `CriterionResult`, `AuditReport`). Reducers (`operator.ior` for evidences, `operator.add` for opinions and errors) are annotated on the state keys that receive parallel updates.
- **Alternatives considered:** (1) Plain `dict` for state — minimal boilerplate but no type checking or schema validation; LangGraph's reducer semantics would be undocumented. (2) A single Pydantic `BaseModel` for the whole state — would give validation and schema but LangGraph expects partial updates; merging parallel updates into one model is awkward and would require custom reducers for every key. (3) Dataclasses for nested types — lighter than Pydantic but no `with_structured_output` integration or JSON schema for LLM outputs.

- **Trade-offs:** TypedDict does not validate at runtime (only static typing); we rely on nodes to write the correct keys. Pydantic adds dependency and serialization rules (e.g. opinions stored as `model_dump()` dicts in state to avoid LangGraph serialization issues). We accepted these to get clear contracts, reducer support for parallel merge, and structured LLM outputs via Pydantic-bound schemas.

Three detectives

- **Choice:** Three separate nodes — RepoInvestigator (code/repo, AST, git), DocAnalyst (PDF, cross-reference), VisionInspector (diagram detection) — each writing into shared state with `operator.ior` (evidences).
- **Alternatives considered:** (1) One detective node that runs all forensic steps in sequence — only one node to maintain and no reducer complexity, but no parallelism and a single point of failure; also mixes concerns (repo, PDF, diagram) in one place. (2) One detective that calls multiple tools in a loop — similar to (1); tool outputs would still need to be merged into state by hand. (3) More than three detectives (e.g. separate node for git history) — finer granularity but more edges and more reducer keys; the rubric's "three detectives" and "parallel" expectations are satisfied with three.
- **Trade-offs:** Three nodes mean three separate repo clones when both RepoInvestigator and DocAnalyst (and VisionInspector) need the repo; we accepted duplicate clone work for clearer separation of roles and independent parallelism. Consolidating to a single "repo fetcher" that runs before the three would require a graph change (e.g. a fourth node that runs first and writes repo path or file list into state); that is a possible future improvement.

Three judges + Chief Justice

- **Choice:** Three judge nodes (Prosecutor, Defense, Tech Lead) and one Chief Justice node that applies deterministic rules (security override, fact supremacy, variance/dissent, functionality weight); no LLM in Chief Justice.
- **Alternatives considered:** (1) Single "grader" LLM that scores all criteria — one LLM call per rubric dimension, lower cost and simpler, but no dialectical tension or rubric-required "distinct perspectives"; single point of bias. (2) Two judges (e.g. Prosecutor and Defense only) — less cost than three but Tech Lead acts as tie-breaker and pragmatic lens; dropping it would make tie-breaking more arbitrary. (3) Chief Justice as another LLM that "synthesizes" the three opinions — would allow natural-language summaries but the verdict would be non-deterministic and harder to audit; the rubric asks for deterministic synthesis.
- **Trade-offs:** Three judges multiply LLM calls (3 × number of dimensions). We accepted the cost to get adversarial vs charitable vs pragmatic viewpoints and to satisfy the rubric. The Chief Justice is pure Python so the final score is reproducible and explainable; we gave up flexible summary text from an LLM for auditability.

Conditional edges

- **Choice:** After the evidence aggregator, a conditional edge routes to `to_judges` (no errors, evidences present) or `handle_failure_or_missing` (errors or empty evidences); both paths then fan out to the same three judges.
- **Alternatives considered:** (1) No branch — on any detective error or missing evidence, abort the run and return an error message. Simpler but the rubric explicitly requires handling "Evidence Missing or Node Failure" without aborting; we would not produce a report. (2) Branch only on "hard" failure (e.g. exception) but not on "empty evidence" — would allow partial reports when one detective fails, but "evidence missing" is a distinct case the rubric calls out. (3) Multiple branches (e.g. different paths for "missing evidence" vs "node failure") — both paths currently do the same thing (append a message to `errors` and proceed to judges); we could split later if we wanted different messaging or behaviour.
- **Trade-offs:** The conditional adds a routing function and two target nodes (`to_judges` is a passthrough). We accepted the extra complexity so the pipeline always reaches the judicial layer and the report can explicitly state that the audit was degraded when evidence was missing or a node failed.

Sandboxing strategy for cloning unknown repositories

The auditor accepts an arbitrary GitHub repository URL. Cloning must be safe against malicious or malformed URLs and must not affect the host's working directory or other assets.

- **Choice:** (1) **Isolated directory:** All clones go into a temporary directory created with `tempfile.TemporaryDirectory()` (or `tempfile.mkdtemp()` when the caller passes a parent dir). The implementation never uses the process current working directory as the clone target. If the caller passes `target_dir` equal to `os.getcwd()`, `_ensure_sandbox_dir()` in `src/tools/git_tools.py` detects this and forces a temp dir instead, so we never clone into the live project. (2) **No shell:** `git clone` is invoked via `subprocess.run(["git", "clone", repo_url, repo_path], ...)` with a list of arguments; no `shell=True` and no string interpolation of the URL into a shell command, which would expose us to injection. (3) **URL validation:** The repo URL is validated with a strict pattern (`REPO_URL_PATTERN`): only `https?://` or `git@host:path` forms are allowed; `file://`, spaces, newlines, and characters like `;`, `|` are rejected before any subprocess call. (4) **Timeout and errors:** Clone runs with `timeout=120`; `CalledProcessError` and `TimeoutExpired` are caught and turned into a `RuntimeError` with a classified message (auth, not found, network, etc.). (5) **Post-clone check:** After clone, we verify the path is a valid git repo (e.g. `.git` exists); if not, we raise rather than returning a non-git directory.
- **Alternatives considered:** (1) Clone into a fixed subdirectory under the project (e.g. `./audit/cloned_repos/`) — would allow reuse across runs but would persist possibly malicious repo content and could mix runs; temp dirs are deleted when the context exits. (2) Use `git clone --depth 1` only — would reduce disk and time but the rubric's git-forensic dimension expects history (e.g. commit progression); we use a full clone. (3) Run clone in a container or VM — strongest isolation but adds

operational cost and is not required for the current threat model (untrusted URL and payload, not untrusted code execution from the cloned repo).

- **Trade-offs:** Each detective that needs the repo (RepoInvestigator, DocAnalyst, VisionInspector) currently performs its own clone inside a `with tempfile.TemporaryDirectory() as tmpdir` block, so we clone the same repo up to three times per run. We accepted duplicate clone work for simplicity and to keep detectives independent and parallel; a future improvement is a shared "repo fetcher" node that clones once and writes the path (or file list) into state. Sandboxing is process-local (temp dir + no shell); we do not run cloned code, so we rely on the OS and filesystem for isolation rather than a sandboxed execution environment.

This rationale is implemented in the codebase and is fully explained in Sections 3, 4, and 6 of this report.

7.2 Gap Analysis and Forward Plan

Resolved gaps (already addressed)

- **Report accuracy (cross-reference):** DocAnalyst now calls `get_repo_file_list(repo_url)` and passes it to `verify_file_claims`; paths normalized; verified vs hallucinated counts based on real repo contents.

- **Safe tool engineering (false positive):** AST-based check counts only actual `os.system` call nodes, not string literals or comments.

- **Structured output (forensic detection):** Uses Pydantic is true when `with_structured_output` and `JudicialOpinion` appear, even if "Pydantic" is not in the judges file.

- **Judicial and Chief Justice evidence depth:** Snippet lengths for judges.py and justice.py (and graph structure) increased so persona prompts, judicial logic, and Chief Justice rules are visible to judges.

- **Architectural diagram criterion:** VisionInspector clones repo and checks `docs/architecture.md` and README for Mermaid (flowchart, graph, stateDiagram); repo includes architecture diagram.

Forward plan for unbuilt components (judicial layer and synthesis engine)

The following focuses on what is **not yet built** for the judicial layer and the synthesis engine, with **specific tasks**, **sequencing**, and **anticipated failure modes**.

A. Judicial layer — unbuilt components

#	Unbuilt component	Description	Specific tasks	Sequencing	Anticipated failure modes
A1	**Evidence-citation validation**	Judges return `cited_evidence` (e.g. UUIDs or keys); the system does not verify that these IDs exist in `state.evidences` before or after synthesis.	(1) Add a validation step in each judge node or in a post-judge step: resolve each `cited_evidence` entry against `state.evidences` and mark invalid/missing refs. (2) Optionally pass validation result (e.g. count of valid vs invalid citations) into Chief Justice or into the report (e.g. "Judge X cited 2 evidence IDs not found in state"). (3) Add rubric dimension or synthesis rule: "If a judge cites non-existent evidence, discount or flag that opinion."	After judges return opinions, before Chief Justice. Validation can run in a small node that reads `state.evidences` and `state.opinions` and appends a `citation_validation` map to state.	Judges may cite UUIDs from a different run or typo IDs; validation returns "all invalid" and could wrongly penalize. Mitigation: normalize evidence IDs (e.g. criterion_id + index) and document ID format in prompts.
A2	**Judge timeout / partial failure**	If one judge (e.g. Prosecutor) times out or raises, the graph may still merge partial opinions; Chief Justice currently assumes exactly three opinions per criterion.	(1) Define contract: each judge returns a list of opinions (one per dimension); on timeout/exception, return empty list or a single "error" opinion with score=1 and argument="Evaluation failed (timeout/error)." (2) In Chief Justice, handle missing	Implement when adding retry/timeout in judge invocation (see A3). Chief Justice change must follow the new contract.	If two judges timeout, final score may rely on one judge only; report could be misleading. Mitigation: require at least two opinions per criterion for a valid score; otherwise mark criterion as "Inconclusive."

#	Unbuilt component	Description	Specific tasks	Sequencing	Anticipated failure modes
			opinions per criterion: if fewer than three, apply rule (e.g. use only available opinions, or set final_score=1 with dissent "Missing judge N"). (3) Add state key `judge_errors: List[str]` and append "Prosecutor timed out" so the report can state partial evaluation.		
A3	**Retry and fallback for judge LLM failures**	Today there is retry for structured-output parse failures; there is no timeout or fallback when the LLM never returns (e.g. API hang).	(1) Wrap judge LLM invoke in a timeout (e.g. `asyncio.wait_for` or thread with timeout). (2) On timeout, append to `judge_errors` and push a single JudicialOpinion per dimension with score=1, argument="Judge evaluation timed out." (3) Optionally: fallback to a lighter model or cached "neutral" opinion for that criterion so Chief Justice always has three opinions.	Can be done in parallel with A2; A2 consumes the new behaviour.	Timeout too short may increase false "timeout" opinions; too long may block the run. Mitigation: make timeout configurable (e.g. from rubric or env) and log timeouts for tuning.
A4	**Rubric-driven persona weighting**	Synthesis rules are hardcoded (e.g. "Tech Lead breaks tie"). The rubric's `synthesis_rules` (e.g. `functional`	(1) Parse `synthesis_rules` from rubric into a small schema (e.g. which dimension types use which tie-breaker, variance threshold). (2) In	After current Chief Justice logic is stable; requires rubric schema extension or convention (e.g. keys like	Malformed or missing rubric keys could fall back to current hardcoded behaviour; document default and validate rubric at load.

#	Unbuilt component	Description	Specific tasks	Sequencing	Anticipated failure modes
		`ity_weight`, `dissent_requirement`) are not read and applied as parameters (e.g. which judge has tie-break for which dimension type).	Chief Justice, use these parameters when applying rules (e.g. for "graph_orchestration" use Tech Lead weight from rubric). (3) Unit test with two different rubrics to ensure behaviour changes with rubric.	`tie_breaker_architecture`).	
A5	**Persona calibration / drift detection**	No mechanism to detect if the three judges collapse to similar scores (persona drift) or if one judge systematically disagrees with evidence.	(1) After each run, compute per-criterion variance and mean score; log or store in report metadata. (2) Optional: if variance is below a threshold for all criteria, add a report note "Low variance: consider reviewing persona prompts for distinctness." (3) Optional: compare cited_evidence overlap between Prosecutor and Defense to detect collusion.	Post-run analysis; can be a separate script or a final node that amends report metadata.	May add noise if rubrics are simple and evidence is clear; make the note optional and threshold configurable.

B. Synthesis engine — unbuilt components

#	Unbuilt component	Description	Specific tasks	Sequencing	Anticipated failure modes
B1	**Rubric-driven synthesis parameters**	Variance threshold (e.g. 2) and security cap (e.g. 3) are hardcoded in Chief Justice. Rubric already has `synthesis_rules` (e.g. `variance_re_evaluation`, `security_override`) but no numeric parameters.	(1) Extend rubric schema or convention: e.g. `variance_threshold: 2`, `security_cap_score: 3`, `dissent_variance_min: 2`. (2) Load these in Chief Justice (from state or rubric); use in conditionals (e.g. `if score_variance > variance_threshold`). (3) If rubric omits them, use current defaults and document.	After A4 if both use rubric; otherwise independent.	Wrong or extreme values (e.g. variance_threshold=0) could force dissent on every criterion; validate and clamp to sane ranges.
B2	**Remediation text beyond Tech Lead copy**	Remediation is currently the Tech Lead argument string (and optional suffix when score < 3). There is no structured, file-level or criterion-level remediation generated from all three opinions.	(1) Define a small RemediationStep model (e.g. file path, suggested change, source judge/criterion). (2) In Chief Justice, for criteria with final_score < 3, optionally invoke a dedicated "remediation summarizer" (LLM or rule-based) that takes Prosecutor, Defense, and Tech Lead arguments and produces a short, ordered list of steps. (3) If LLM: keep it optional and behind a flag so deterministic report path remains default.	After Chief Justice outputs CriterionResult; can be a second pass over criteria_results.	LLM-based remediation could hallucinate file paths; prefer rule-based extraction (e.g. "add reducer" from Prosecutor argument) or restrict LLM to summarization only.
B3	**Executive summary enrichment**	Executive summary is template-built (scores,	(1) Option A: Add an optional LLM pass that takes executive_summary +	Post Chief Justice; can run before or after B2.	LLM narrative could contradict scores; use as supplement only

#	Unbuilt component	Description	Specific tasks	Sequencing	Anticipated failure modes
		counts). No narrative nuance (e.g. “Strong in state management; weak in report accuracy”).	criteria_results and returns 2–3 sentence narrative; append to report. Option B: Rule-based: “Strengths: criteria with score ≥ 4; Weaknesses: criteria with score < 3” and list dimension names. (2) Keep current template as fallback if enrichment is disabled or fails.		and keep template summary as canonical.
B 4	**Dissent summary enrichment**	Dissent summary is built from truncated Prosecutor/Defense arguments and resolution rationale. No structured “conflict type” (e.g. evidence vs interpretation) or link to synthesis rule applied.	(1) In Chief Justice, when building dissent_summary, append the name of the rule that resolved the tie (e.g. “fact_supremacy”, “tech_lead_tie_breaker”). (2) Optionally classify dissent (e.g. “evidence_quality” vs “interpretation”) from keywords in arguments. (3) Expose in report so readers see which synthesis rule was used.	Can be done inside current Chief Justice loop when variance > 2.	Classification might be wrong; keep it optional and descriptive only.
B 5	**Export to other formats**	Report is Markdown and JSON only. No HTML, PDF, or schema-validated JSON (e.g. JSON Schema) for downstream tools.	(1) Add serializer for HTML (e.g. Jinja2 template or Markdown-to-HTML). (2) Optionally add PDF export (e.g. weasyprint or reportlab). (3) Publish JSON Schema for audit_report.json and validate before write.	Independent; can be parallel to other items.	PDF/HTML may have styling or encoding issues; validate on multiple samples.

C. Sequencing and dependencies

2. **Phase 1 (robustness):** A2 (missing-opinion handling) and A3 (timeout/fallback) so the pipeline tolerates judge failures.

3. **Phase 2 (correctness):** A1 (citation validation) and B4 (dissent enrichment) so the report reflects real evidence and applied rules.

4. **Phase 3 (configurability):** A4 (rubric-driven persona weighting) and B1 (rubric-driven synthesis parameters) so behaviour is driven by rubric.

5. **Phase 4 (quality):** B2 (remediation), B3 (executive summary), A5 (persona drift) as optional enhancements.

6. **Phase 5 (output):** B5 (export formats) as needed.

This forward plan addresses the rubric feedback that the plan for *unbuilt* judicial and synthesis components should be more granular, with specific tasks, sequencing, and anticipated failure modes.

8. Conclusion

Automaton Auditor implements the Week 2 challenge through a structured, multi-agent "Digital Courtroom" architecture. The system collects forensic evidence in parallel via RepoInvestigator, DocAnalyst, and VisionInspector; applies adversarial judicial reasoning via Prosecutor, Defense, and Tech Lead with Pydantic-bound JudicialOpinion output; and enforces deterministic synthesis in the Chief Justice node using hardcoded rules, producing a single AuditReport. It produces traceable, auditable reports in Markdown and JSON, with optional comparison to a peer report.

The architecture is explicitly modeled as a LangGraph StateGraph with clear parallelism, safe state merging (operator.ior, operator.add), and conditional routing for robustness. This report contains the full rationale, execution flow, project structure, **StateGraph diagrams with state types on edges and explicit conditional/error-handling paths**, and an **expanded gap analysis and forward plan for unbuilt judicial and synthesis components** (specific tasks, sequencing, and failure modes). The submission includes source code, machine-readable rubric, architecture diagrams (in this report and in the repo), structured reports, tests, and the gap analysis and forward plan described in Section 7.2.