Automaton Auditor–Interim Report

1. Architecture decisions

Why Pydantic over dicts ?
LangGraph passes a single state object between nodes. If that state were a plain Python dict, parallel nodes such as RepoInvestigator and DocAnalyst could overwrite each other's keys, so one detective's evidence could replace the other's. We therefore define shared state with TypedDict and Pydantic models: `Evidence`, `JudicialOpinion`, `CriterionResult`, and `AuditReport` are Pydantic `BaseModel` classes, and `AgentState` is a TypedDict. This gives a clear schema and validation for tools and downstream nodes. For fields that multiple nodes write to, we use reducers: `evidences` is annotated with `operator.ior` so that each detective's output is merged into the dict instead of replacing it, and `opinions` uses `operator.add` so that each judge appends to the list. That merge behaviour cannot be expressed with untyped dicts alone, so Pydantic and TypedDict are necessary for correct parallel behaviour.

How was AST parsing structured ?
The goal is to determine whether the target repository actually uses a StateGraph with real nodes and edges (including fan-out and fan-in) without relying on regex, which is brittle for code structure. In `src/tools/repo_tools.py`, `analyze_graph_structure(path)` reads `src/graph.py` from the cloned repo and uses Python's built-in `ast` module to parse it. We walk the AST looking for `StateGraph` usage, `add_edge` and `add_conditional_edges` calls, and `add_node` calls (from which we collect node names). From that we derive booleans such as `has_state_graph`, `parallel_fan_out`, and `has_sync_node`, and we attach a short code snippet to the evidence. No regex is used for graph topology. For state definitions we use `analyze_state_structure(path)`, which parses `src/state.py` with the same AST approach to detect BaseModel, TypedDict, Evidence, JudicialOpinion, and the use of `operator.add` and `operator.ior` as reducers.

Sandboxing strategy
All git operations run in an isolated temporary directory. The function `clone_repo_sandboxed(repo_url)` is a context manager that creates a `tempfile.TemporaryDirectory()`, runs `git clone` via `subprocess.run()` (with captured stdout/stderr and return-code checks), and yields the clone path. When the context exits, the temporary directory is removed, so the live working directory is never used for clones and no repo content is left on disk. We do not use `os.system()` for clone or git commands; subprocess is used so that we can handle authentication and clone failures and surface them as clear errors (e.g. `PermissionError` for auth failure). This keeps the auditor safe when cloning arbitrary repositories.

2. Known gaps

The current implementation stops at the Detective layer. The Prosecutor, Defense, and Tech Lead judge nodes are not implemented, so there is no scoring or dialectical synthesis. The Chief Justice synthesis engine is also missing: there are no hardcoded conflict-resolution rules and no final `AuditReport` or Markdown report generation. VisionInspector (diagram analysis from PDF images) is not implemented and is out of scope for this interim. The graph has no conditional edges for “Evidence Missing” or “Node Failure”; those are planned for the final submission.

- Judicial layer: Prosecutor, Defense, and Tech Lead nodes are not implemented. The graph stops at EvidenceAggregator; no scoring or dialectical synthesis.
- Chief Justice: No synthesis engine or hardcoded conflict-resolution rules. No final `AuditReport` or Markdown report generation.
- VisionInspector: Not implemented. Diagram analysis (PDF images) is out of scope for the interim.
- Conditional edges: The current graph has no conditional edges for “Evidence Missing” or “Node Failure”; those are planned for the final submission.

What is fully working (Detective layer – 100% complete)**
- Sandboxed cloning
- Parallel fan-out to RepoInvestigator + DocAnalyst
- Deep AST-based graph & state analysis (no regex)
- Evidence aggregation with correct reducers
- Full typed state with Pydantic validation
- Clean error handling and logging

What is still missing (gaps)**
- Judicial layer (Prosecutor, Defense, and Tech Lead nodes)
- Any scoring or debate/synthesis logic
- Chief Justice synthesis engine and final report generation
- Conditional routing edges (“Evidence Missing”, “Node Failure”, “Re-evaluate”)
- VisionInspector (PDF diagram image analysis) – explicitly out of scope for this interim

Concrete plan I will execute for the judicial layer
I will create `src/nodes/judges.py` with three dedicated judge nodes:

- Prosecutor – adversarial / strict persona
- Defense – forgiving / benefit-of-doubt persona
- Tech Lead – pragmatic / architecture-focused persona

Each node will:
- Receive the exact same aggregated evidence + rubric dimensions
- Call the LLM using its own distinct system prompt (taken word-for-word from `rubric.json` judicial guidelines)
- Use `.with_structured_output(JudicialOpinion)` so the output is **guaranteed** to match the Pydantic schema (score 0–5, justification, quotes, etc.)

Graph wiring: EvidenceAggregator → parallel fan-out to the three judges → fan-in to ChiefJustice.

3. Concrete plan for judicial layer and synthesis engine

I will add `src/nodes/judges.py` with three nodes—Prosecutor, Defense, and Tech Lead—each receiving the same aggregated evidence and rubric dimensions. Each judge will invoke an LLM with a distinct system prompt (adversarial, forgiving, or pragmatic) aligned to the rubric's judicial guidelines, and we will use `.with_structured_output(JudicialOpinion)` (or `.bind_tools()`) so that every judge output is validated against the Pydantic `JudicialOpinion` schema. The graph will fan out from EvidenceAggregator to these three judges in parallel and then fan in to the Chief Justice node.

I will add `src/nodes/justice.py` with a `ChiefJusticeNode` that takes all `JudicialOpinion` objects per criterion and applies hardcoded Python rules: for example, a security override (if the Prosecutor identifies a confirmed vulnerability, the score is capped at 3 regardless of the Defense), fact supremacy (if the Defense claims something that the Detective evidence contradicts, the Defense is overruled), and functionality weight (the Tech Lead's assessment of architectural modularity carries the highest weight for the architecture criterion). When score variance across the three judges exceeds 2, we will trigger a dissent summary and optional re-evaluation. The node will produce an `AuditReport` and serialize it to Markdown with an Executive Summary, Criterion Breakdown (including dissent where applicable), and Remediation Plan. The `synthesis_rules` from `rubric.json` will be loaded and used in the Chief Justice context.

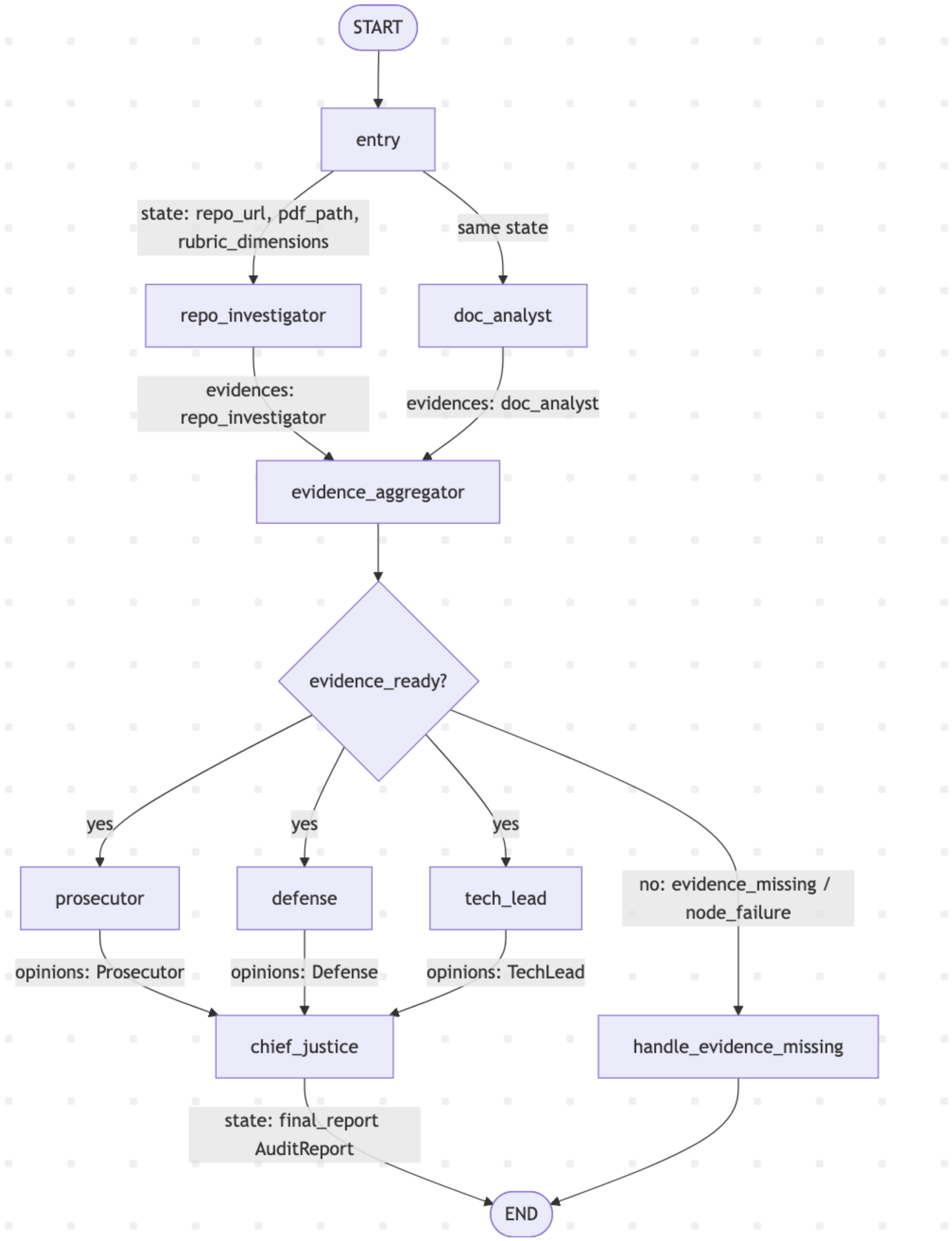
- Judges: Add `src/nodes/judges.py` with three nodes (Prosecutor, Defense, Tech Lead). Each receives the same aggregated evidence and rubric dimensions. LLMs are invoked with `.with_structured_output(JudicialOpinion)` (or `.bind_tools()`) so output is always validated Pydantic. Personas will use distinct system prompts (adversarial / forgiving / pragmatic) from the rubric's judicial guidelines.

- Chief Justice: Add `src/nodes/justice.py` with `ChiefJusticeNode`. Input: all `JudicialOpinion` objects per criterion. Logic: hardcoded Python rules—e.g. security override (Prosecutor-identified vulnerability caps score at 3), fact supremacy (Defense overruled if evidence contradicts), functionality weight (Tech Lead's "modular" assessment weighted highly for architecture). If score variance > 2, trigger dissent summary and optional re-evaluation. Output: `AuditReport` serialized to Markdown (Executive Summary → Criterion Breakdown → Remediation Plan).
- Graph wiring: After EvidenceAggregator, fan-out to the three judges (parallel), then fan-in to ChiefJustice, then END. Load `synthesis_rules` from `rubric.json` into the Chief Justice context.

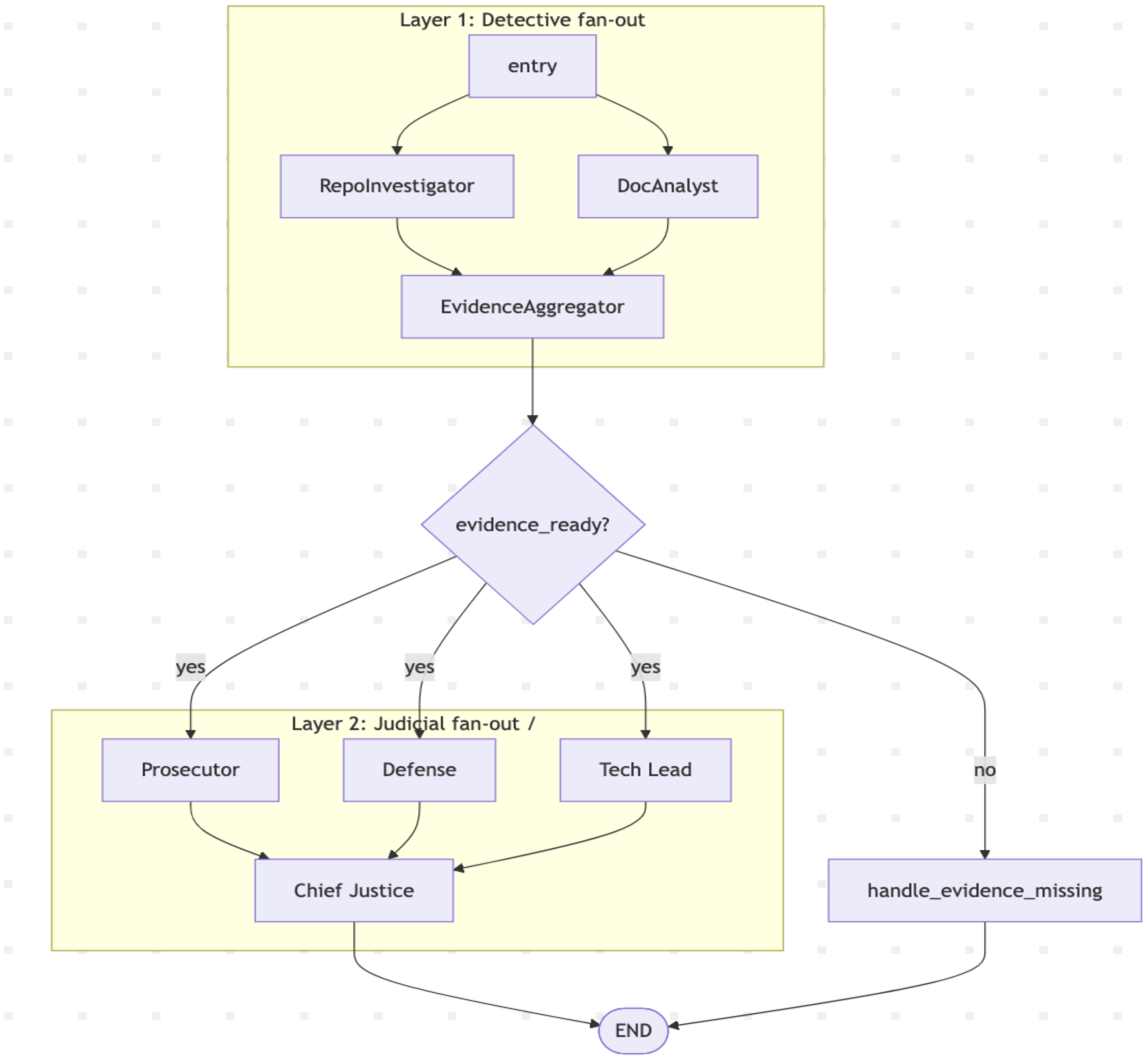
4. StateGraph Orchestration Flow

Full planned architecture (entire flow)

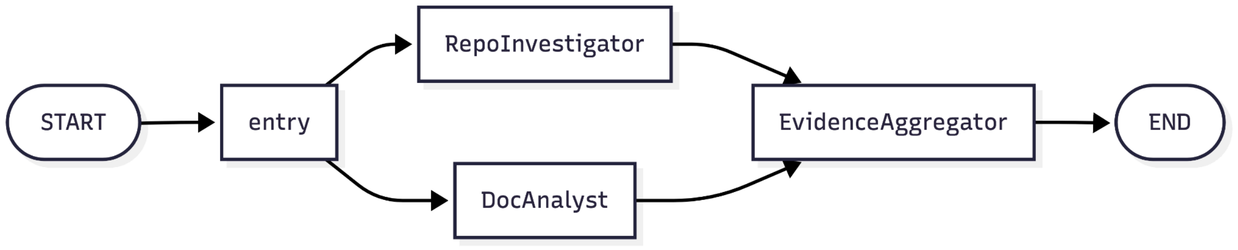
The diagram below shows the **entire** planned StateGraph: Detective layer (implemented), Judicial layer (planned), and conditional edges for error handling. Components not yet built are labeled for clarity.Current partial flow (implemented): Entry sets `repo_url`, `pdf_path`, and `rubric_dimensions`. The graph then fans out to RepoInvestigator and DocAnalyst, which run in parallel. Both feed into EvidenceAggregator (fan-in), which performs cross-reference and prepares state for future judges. The run ends there; no Judge or Chief Justice nodes are present yet.


START
entry
state: repo_url, pdf_path, rubric_dimensions
same state
repo_investigator
doc_analyst
evidences: repo_investigator
evidences: doc_analyst
evidence_aggregator
evidence_ready?
yes
yes
yes
no: evidence_missing / node_failure
prosecutor
defense
tech_lead
opinions: Prosecutor
opinions: Defense
opinions: TechLead
chief_justice
handle_evidence_missing
state: final_report AuditReport
END

Visual summary: two fan-out/fan-in layers + conditional



Partial StateGraph orchestration



1. Entry nodes populate the initial state.
2. Parallel execution of two detectives (RepoInvestigator uses sandboxed clone + AST; DocAnalyst reads the PDF).
3. EvidenceAggregator merges everything safely using the reducers I defined.
4. Graph ends here for the interim submission (no judges yet).