

Interim Submission

Automaton Auditor

Hiwot Beyene

Feb 25 , 2026

1. Architecture Decisions Made So Far

1.1 Why Pydantic Over Dicts

I decided that all state and agent outputs should use **Pydantic BaseModel** for structured data and **TypedDict** for graph state. Plain Python dicts are not used for **AgentState** or for Detective or Judge outputs.

1.1.1 Why did I chose this?

1. LangGraph state reducers:

- operator.ior used for evidences (combines sets/union-style)
- operator.add used for opinions (concatenates/appends)

These reducers require consistent types across parallel nodes → merging works correctly only when types match.

2. Pydantic models are used (instead of plain dicts) because:

- They serialize/deserialize cleanly
- They prevent overwrites with arbitrary/inconsistent dict shapes
- They keep parallelism safe and predictable

3. Judges must return structured JSON with fields: **score**, **argument**, **cited_evidence**

- Implementation uses **.with_structured_output(JudicialOpinion)**
- Forces LLM to follow the schema → free-text or malformed output → parse error → retry
- Plain dicts would allow “dict soups” and hallucinated keys

4. Detectives (for forensic evidence) emit **Evidence** objects with typed fields:

- goal
- found
- content
- location
- rationale
- confidence (validated in [0, 1]) → Keeps evidence machine-readable and auditable

5. Tech Lead statute in the rubric explicitly states:

- "Pydantic Rigor vs. Dict Soups" rule applies
- State and JSON outputs **must** use typed structures
- Otherwise → ruling is "**Technical Debt**" (**Score 3**)

1.1.2 What I implemented ?

1. In `src/state.py`, the following classes are defined as Pydantic BaseModel:

- Evidence
- JudicialOpinion
- CriterionResult
- AuditReport

2. The graph state `AgentState` is defined as a `TypedDict`. The reducer keys in `AgentState` use:

- `Annotated[dict[str, list[Evidence]], operator.ior]`
- `Annotated[list[JudicialOpinion], operator.add]`

1.2 How AST Parsing Was Structured

I decided that RepoInvestigator would use **Python's built-in ast module** in `src/tools/repo_tools.py`. I do not use regex to infer code structure (for example, to find `StateGraph`, `add_edge`, or reducers).

Why did I chose this?

The challenge's Forensic Protocol B (Graph Wiring) requires verifying that the graph is *functionally* wired for parallelism.

for example, that `builder.add_edge()` creates fan-out and that a sync node handles fan-in. Regex can match strings without confirming valid Python syntax; the AST confirms both syntax and structure. Forensic Protocol A (State) requires detecting classes that inherit from `BaseModel` or `TypedDict` and the use of `Annotated` with `operator.ior` or `operator.add`. The AST allows precise detection of class bases and type annotations. The same AST-based approach can later be extended to inspect `src/tools/` (for tempfile, subprocess, and the absence of `os.system`) and `src/nodes/judges.py` (for `.with_structured_output()` bound to `JudicialOpinion`) once those exist.

How it works in practice?

The function `analyze_graph_structure(repo_path)` parses `src/graph.py` and `src/state.py` under the cloned repo path. For graph topology, I walk the AST for `Call` nodes and detect `add_edge(source, target)` and `add_conditional_edges(source, ...)` on a builder, and I collect `add_node(name)`.

The function returns `nodes`, `edges`, `has_state_graph`, and `has_conditional_edges`. For state and reducers, I look at classes whose bases include `TypedDict` or `BaseModel`, scan their `AnnAssign` annotations for `Annotated[...]`, and collect `operator.ior` and `operator.add` from nested `Attribute` nodes. The function returns `state_classes` and `reducers`. I do not use regex for code structure; all inference comes from the AST. This keeps evidence objective and aligned with the rubric's "Deep AST Parsing" standard.

1.3 Sandboxing Strategy

I decided that all git operations would run inside a **sandboxed temporary directory** created with `tempfile.TemporaryDirectory()`. I use `subprocess.run` with a timeout and with stdout and stderr capture. I do not use `os.system()` or pass an unsanitized URL into a shell.

Why I chose this?

The agent clones *unknown* repositories. Cloning into the live working directory would allow arbitrary code to affect the auditor process. The rubric requires a sandboxed clone only. The Statute of Engineering in the rubric states that raw `os.system('git clone <url>')` with unsanitized input counts as "Security Negligence" and overrides effort points for Forensic Accuracy. Using subprocess with return-code and stderr capture lets us handle authentication failures, "Repository not found", and invalid hosts in a precise way; I turn those into a `RepoCloneError` with clear messages and then into `Evidence(found=False, rationale=...)` instead of letting uncaught exceptions propagate.

What I implemented in src/tools/repo_tools.py. ?

The context manager `sandboxed_clone(repo_url)` creates a `tempfile.TemporaryDirectory(prefix="auditor_clone_")`, runs `git clone --depth 50 <url> <tmp.name>` via `subprocess.run(..., capture_output=True, text=True, timeout=CLONE_TIMEOUT_SEC)`, and yields the clone path; on exit, the temp directory is removed. I sanitize the URL by rejecting an empty URL, disallowing characters (newline, space), and invalid patterns; I recognize `https://...` and `git@...` and raise `RepoCloneError` with specific messages for "Git authentication failed", "Repository not found or inaccessible", and "Invalid or unreachable host".

The function `extract_git_history(repo_path)` uses `subprocess.run(["git", "log", ...], capture_output=True, timeout=GIT_LOG_TIMEOUT_SEC, cwd=repo_path)` and returns a list of `{message, timestamp}`. For DocAnalyst and VisionInspector, the PDF path comes from state and file reads are local and path-controlled; I do not execute shell commands there.

1.4 Error Handling: Skip as Primary, Retry Only for Transient Failures

I decided that skip is the primary path for Evidence Missing and Node Failure:

- when evidence is incomplete or a node has failed, I do not invoke the Judges and instead follow the skip branch to END (or to an "Error: Missing Evidence" outcome)
- .Retry is used only at the operation level for transient failures, not as a graph-level "Retry?" node.

Why I chose this?

conditional edges is required for Evidence Missing and Node Failure. Fail-fast on missing or invalid evidence avoids wasting LLM calls and keeps the graph simple. Retrying at the graph level (e.g. re-running a detective) would complicate state and could mask permanent failures. Transient failures (e.g. clone timeout, temporary network error) are different: they can succeed on a later attempt, so I retry only inside the operation that failed.

What I implemented?

In `sandboxed_clone`, I retry only on transient conditions: timeout and connection-related stderr. I use bounded retries (`MAX_CLONE_RETRIES`) and exponential backoff. Permanent errors (authentication failure, repository not found, invalid URL) raise `RepoCloneError`

immediately with no retry. After all retries are exhausted I still raise; the detective then returns evidence with found=False and the graph's conditional edge routes to "skip" and END. So the graph always sees either success (proceed) or failure (skip); there is no graph-level retry node. The src/graph.py docstring documents this: "Skip is the primary path for missing evidence; transient failures are retried with backoff inside sandboxed_clone; after retries exhausted I still skip."

2. Known Gaps and Concrete Plan for Judicial Layer and Synthesis Engine

2.1 Known Gaps

The graph currently ends after the EvidenceAggregator, with a conditional proceed or skip leading to END. I do not yet have any nodes in src/nodes/judges.py (Prosecutor, Defense, TechLead) or edges from the aggregator to those judges.

The rubric dimensions, including forensic_instruction and judicial_logic, live in rubric.json and are loaded by the API, and the detectives already filter by target_artifact. However, I do not yet invoke Judges per criterion with judicial_logic in their prompts. I also do not have src/nodes/justice.py with a ChiefJusticeNode that implements hardcoded conflict-resolution rules (security override, fact supremacy, functionality light, and dissent when variance is greater than 2).

The Judges are not implemented yet;

- when I add them, they must use .with_structured_output(JudicialOpinion) or .bind_tools() and retry when the model returns free text.
- The AuditReport type exists in the state schema but I do not yet produce or serialize it to Markdown, and I do not write any report to reports/ or audit/.
- The VisionInspector node exists and extracts images from the PDF, but I do not invoke the vision model for analysis in this interim scope.

What I have already implemented ?

- The module `src/tools/repo_tools.py` implements `sandboxed_clone`, `extract_git_history`, and `analyze_graph_structure` (AST-based). RepolInvestigator uses them and raises `RepoCloneError` for bad URL or auth failures. The `evidence_aggregator` node uses

`add_conditional_edges(_route_after_aggregator, {"proceed": END, "skip": END})` so that I can handle missing evidence or skip unavailable artifacts.

- The file `rubric.json` at the repo root is loaded by the API, which exposes `GET /api/rubric`; the detectives receive dimensions filtered by `target_artifact`. The run command for the detective graph is: `uv run python scripts/run_audit.py <repo_url> [pdf_path]`.

2.2 Concrete Plan: Judicial Layer (Phase 3)

I will add `src/nodes/judges.py` with three node functions: Prosecutor, Defense, and TechLead. Each node will receive the aggregated evidence for one criterion plus that criterion's `forensic_instruction` or judicial logic from the rubric. I will call the LLM with `.with_structured_output(JudicialOpinion)`; on parse failure I will retry or return a default low-confidence opinion. The system prompts will be distinct: the Prosecutor will be critical, the Defense will emphasize effort and intent, and the Tech Lead will be pragmatic and act as the tie-breaker.

I already load `rubric.json` at runtime via the API. I will inject `judicial_logic` (or dimension-specific text) into each judge's prompt per dimension. For graph wiring, after the `EvidenceAggregator`'s conditional branch I will route "proceed" to a fan-out to Prosecutor, Defense, and TechLead (either parallel per criterion or via a single `JudicialPanel` node that loops), then fan-in to `ChiefJusticeNode`; on "skip" I will keep the current path to END. The attachment point is already documented in the `src/graph.py` docstring: `proceed` leads to `[prosecutor, defense, tech_lead]` and then to `chief_justice` and `END`. The state already has opinions: `Annotated[list[JudicialOpinion], operator.add]`, so the judges will simply append to it.

2.3 Concrete Plan: Synthesis Engine (Phase 4)

I will add `src/nodes/justice.py` with a `ChiefJusticeNode`. It will take as input `state["opinions"]` (grouped by `criterion_id`), `state["evidences"]`, and the `synthesis_rules` from the rubric. The logic will be pure Python (no LLM). For each criterion I will apply: the Rule of Security (if the Prosecutor flags a confirmed security flaw, cap the score at 3); the Rule of Evidence (if the Defense claims something that the evidence does not support, overrule for hallucination); the Rule of Functionality (if the Tech Lead confirms the architecture is modular, give that the highest light for the architecture criterion); and when score variance is greater than 2 I will set `dissent_summary`. I may optionally re-evaluate the cited evidence before assigning the final score. The node will build an `AuditReport` and assign it to `state["final_report"]`.

I will serialize the AuditReport to Markdown in the required structure (Executive Summary, then Criterion Breakdown, then Remediation Plan) and write it to a file under reports/ or audit/ as specified in the deliverables. The graph will have an edge from the judicial fan-in node to ChiefJusticeNode and then to END. The conditional "skip" branch already routes to END when evidence is missing.

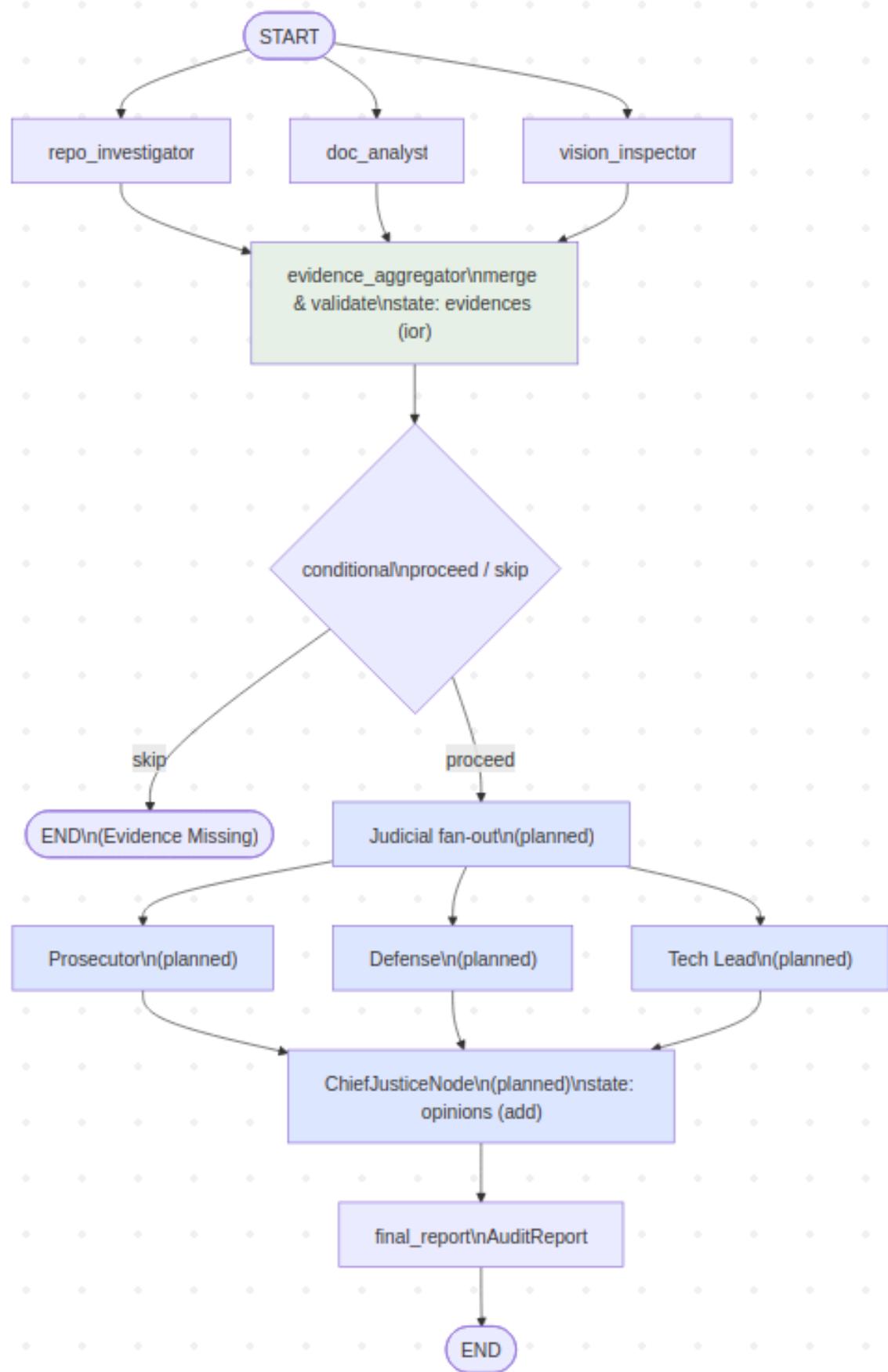
3. StateGraph Flow: Detective Fan-Out and Fan-In

Conditional edges is required for error handling. The rubric (Graph Orchestration Architecture) states that the graph must include "conditional edges that handle 'Evidence Missing' or 'Node Failure' scenarios." Our flow branches after evidence aggregation: when evidence is complete I proceed to the Judicial layer; when evidence is missing or a node has failed I follow the skip path to END (Error: Missing Evidence) and do not invoke the Judges. Retry is not a graph-level node; transient failures (e.g. clone timeout) are retried with backoff inside operations such as sandboxed_clone; after retries are exhausted I still take the skip path.

To meet the rubric's requirement that the diagram depict the entire planned architecture, I show below the full StateGraph: detective layer parallel fan-out/fan-in, the aggregation node with state labels, conditional edges (proceed / skip), and the judicial layer's parallel fan-out/fan-in and synthesis node, even though the judicial components are not yet built. Implemented nodes are in green; planned nodes are in blue with a "(planned)" label.

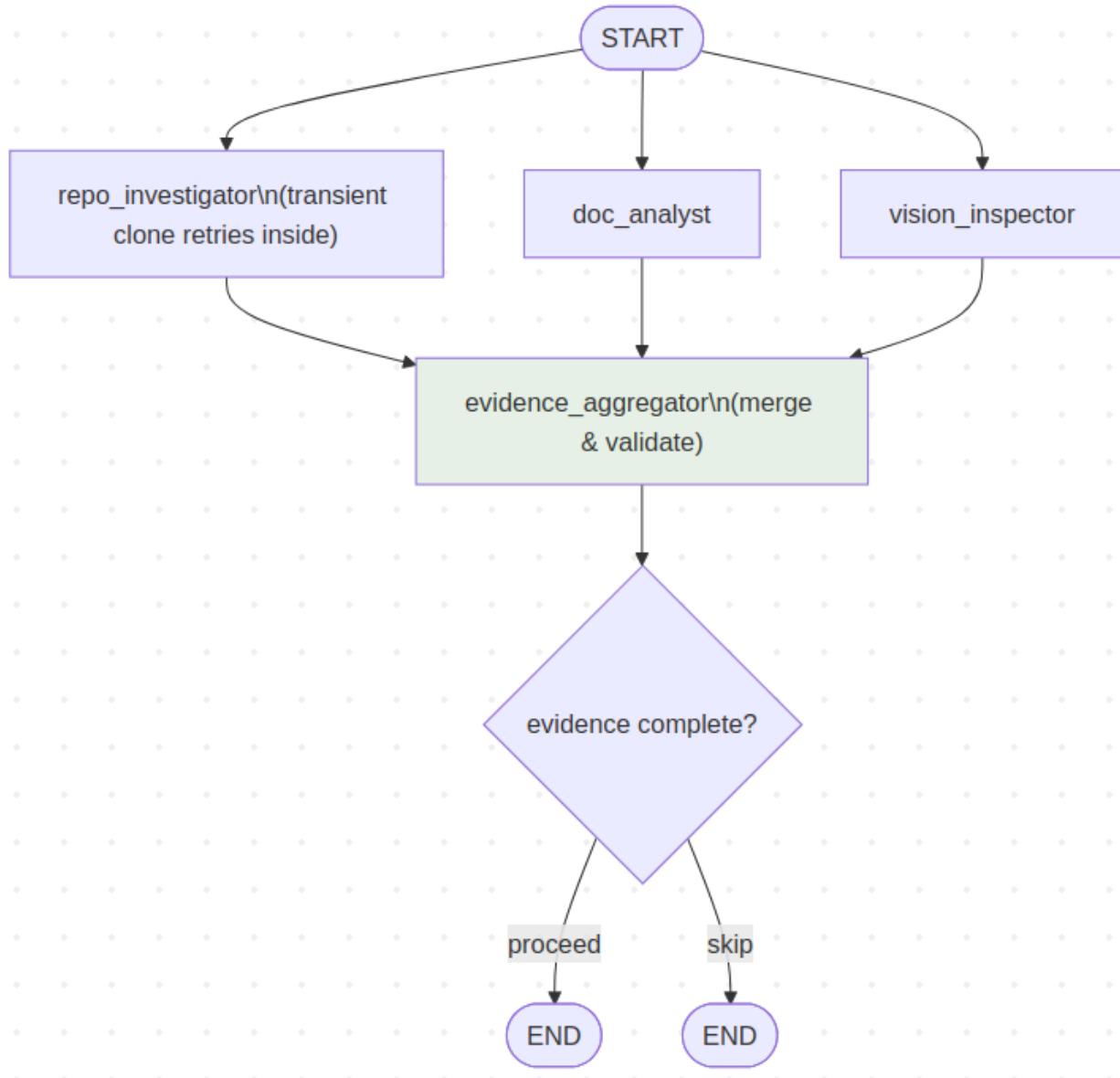
3.1 Full Planned StateGraph Architecture (Diagram)

This diagram visually depicts the complete planned architecture, including the judicial layer's parallel fan-out/fan-in and conditional edges. The Detective Layer (implemented) fans out from START to RepolInvestigator, DocAnalyst, and VisionInspector; they fan in at EvidenceAggregator, which merges and validates evidence (state: evidences, reducer operator.ior). A conditional edge then routes: proceed → Judicial layer fan-out to Prosecutor, Defense, and Tech Lead (planned); they fan in at ChiefJusticeNode (planned), which writes final_report and goes to END. skip → END (Evidence Missing / Node Failure). State keys are indicated at each stage.



3.2 Current Implementation (Partial)

In the current build, only the Detective Layer and EvidenceAggregator are implemented. The conditional edge exists: "proceed" and "skip" both route to END until the judicial layer is wired; "skip" implements the required error-handling path so we never invoke Judges on incomplete evidence. The diagram below shows the implemented partial flow.



3.3 Data Flow Summary

The table below summarizes the stages. In stage 1, START fans out to repo_investigator, doc_analyst, and vision_inspector, and they all fan in at evidence_aggregator; the state key is `evidences` with reducer `operator.ior`. In stage 2, evidence_aggregator uses a conditional (proceed or skip), and on proceed I will route to the Judges (planned). In stage 3 (planned), I will fan out to Prosecutor, Defense, and TechLead per criterion and fan in at ChiefJusticeNode; the state key will be `opinions` with reducer `operator.add`. In stage 4 (planned), ChiefJusticeNode will write `final_report` (an `AuditReport`) and then I reach END.

Stage	Fan-out	Fan-in	State keys
1 (implemented)	START → repo_investigator, doc_analyst, vision_inspector	All → evidence_aggregator	<code>evidences</code> (reducer: <code>operator.ior</code>)
2 (implemented)	evidence_aggregator → conditional (proceed / skip)	skip → END	—
3 (planned)	proceed → Prosecutor, Defense, TechLead (per criterion)	All → ChiefJusticeNode	<code>opinions</code> (reducer: <code>operator.add</code>)
4 (planned)	—	ChiefJusticeNode → END	<code>final_report</code> (<code>AuditReport</code>)