Automaton Auditor — Final Report (Week 2)

1) Executive Summary (What I built, why it matters, and what the system proved)

This project is an **AI-driven auditor** that evaluates a target repository and an architectural PDF report using a "Digital Courtroom" model. Instead of producing a single opinion, the system runs multiple specialized "detectives" to extract **grounded evidence**, then sends that evidence to a panel of "judges" (Prosecutor, Defense, Tech Lead) who debate the quality of the submission under a rubric. Finally, a deterministic "Chief Justice" node synthesizes the judges' opinions into a structured final audit report.

The core value of the system is that it **reduces hallucination risk** by forcing the pipeline to operate on **explicit evidence objects**. Evidence is collected from two places: the code repository (AST/static checks and repository inspection) and the PDF report (semantic chunking and keyword-based forensic search). Judges are instructed to grade **only based on the evidence provided**, and the Chief Justice then computes the final criterion scores in a transparent way.

In testing the system on my own repository (`--mode self`), the pipeline successfully produced:

- deterministic **repo verification** showing LangGraph fan-out/fan-in structure and Typed State reducers,
- PDF chunk extraction and concept matching,
- judge opinions aligned to rubric criterion IDs,
- a final `AuditReport` saved under `audit/report_onself_generated`.

This final report explains the complete architecture, how evidence is generated, how scoring happens, and what I improved through the iterative MinMax feedback loop.

2) Architecture Deep Dive + Diagrams (How the whole system works end-to-end)

2.1 System model: Digital Courtroom pipeline

The system is implemented as a **LangGraph StateGraph** built on a single shared `AgentState`. The graph has three layers:

1. **Detective layer** (parallel fan-out)
    - Repo Detective: clones the repo safely into a sandbox, runs AST/static checks, flags unsafe execution patterns, verifies graph/state architecture.
    - Doc Detective: ingests the PDF using a converter, exports to markdown, creates semantic chunks, and searches for key architecture concepts.
    - Vision Inspector: scans repository assets for architecture diagrams/images and produces evidence of existing visuals.
2. **Judicial bench** (parallel fan-out)
    - Prosecutor: strict and skeptical
    - Defense: generous and credit-giving
    - Tech Lead: engineering-focused and practical
      Each judge outputs a **structured** `JudicialOpinion` object per criterion.
3. **Chief Justice synthesis**
    - Groups judge opinions by criterion.
    - Computes final criterion scores using deterministic logic.
    - Generates an `AuditReport` and writes a markdown report artifact.

This structure matters because it creates **separation of concerns**:

- Detectives do *fact finding* (evidence).
- Judges do *interpretation* (opinions).
- Chief Justice does *aggregation* (final decision).

2.2 State design: Typed state + reducers (why it's reliable)

The project uses a TypedDict-based `AgentState` so every node reads/writes predictable fields:

- `repo_url`, `pdf_path`: inputs
- `evidences`: dictionary of evidence lists keyed by node name

- `opinions`: list of judge outputs
- `final_report`: the output object

Crucially, I used **reducers** so parallel nodes don't overwrite each other:

- Evidence reducer: `operator.ior` merges dictionaries (`{"repo_detective":[...]} + {"doc_detective":[...]}`)
- Opinions reducer: `operator.add` concatenates lists across judges

This ensures the system works correctly under parallel fan-out (multiple nodes running), because the outputs safely merge instead of competing.

2.3 Evidence generation: what counts as "proof" in this system?

Evidence is the foundation of the audit. Every evidence item is a strict Pydantic model:

- goal: what we're trying to prove
- found: True/False
- content: snippet/summary
- location: file path or chunk reference
- rationale: why this evidence supports the goal
- confidence: 0.0–1.0

Evidence is produced by:

- static AST checks (repo_tools)
- repository scans (security patterns, diagram discovery)
- PDF semantic chunking and targeted search (doc_tools)

This design makes the whole system auditable: you can trace every claim back to a location.

2.4 Diagrams (Mermaid syntax)

Diagram A — Full LangGraph pipeline (Detectives → Judges → Chief Justice)

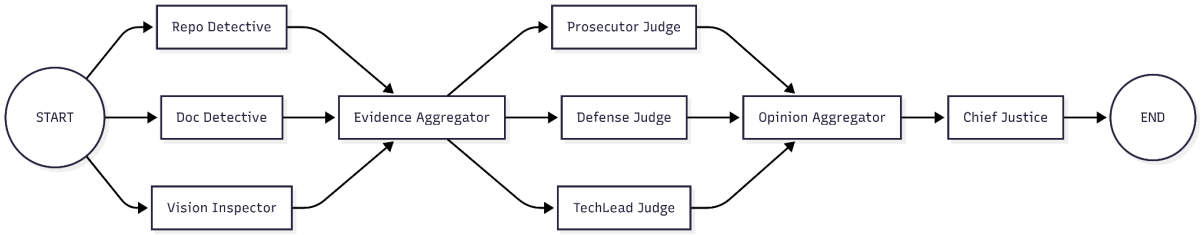


Diagram B — State + reducers (how parallel nodes merge outputs safely)

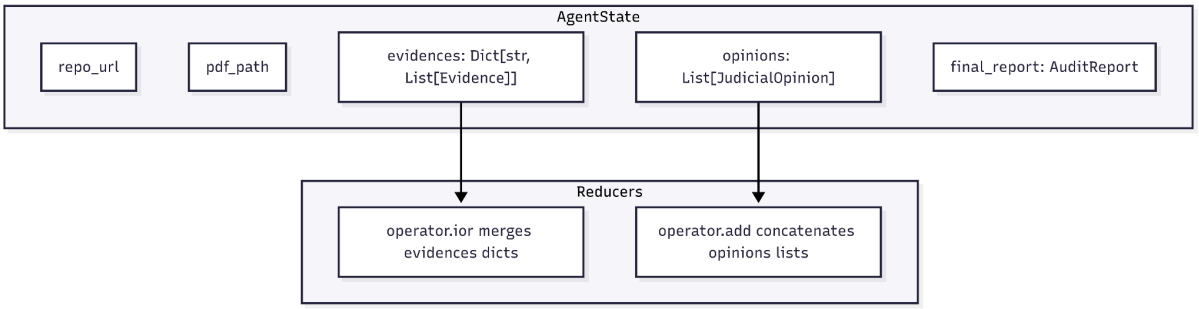


Diagram C — "Evidence → Opinion → Synthesis" responsibility split

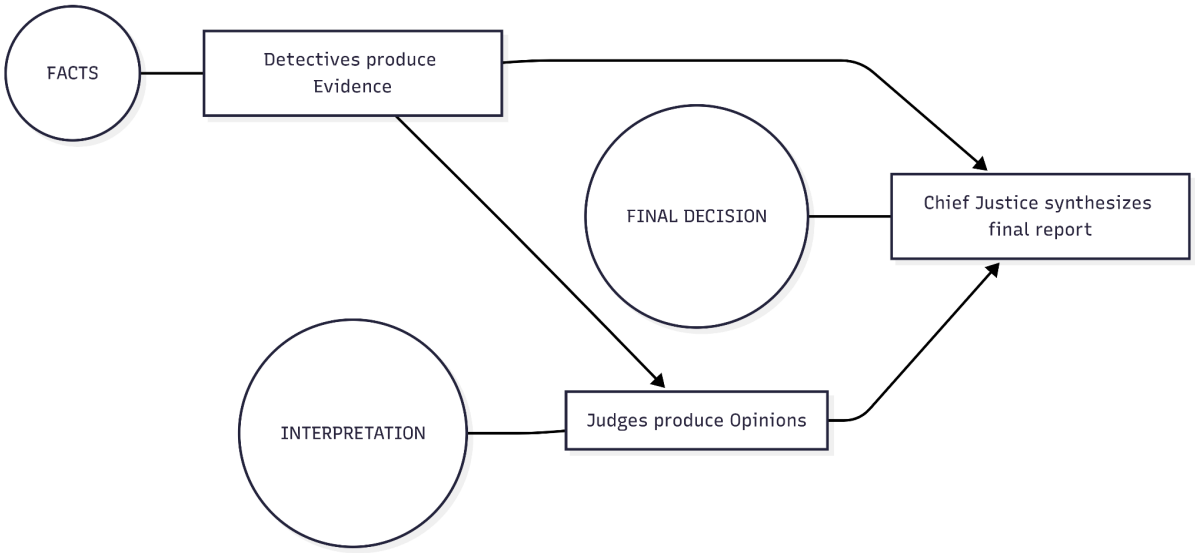
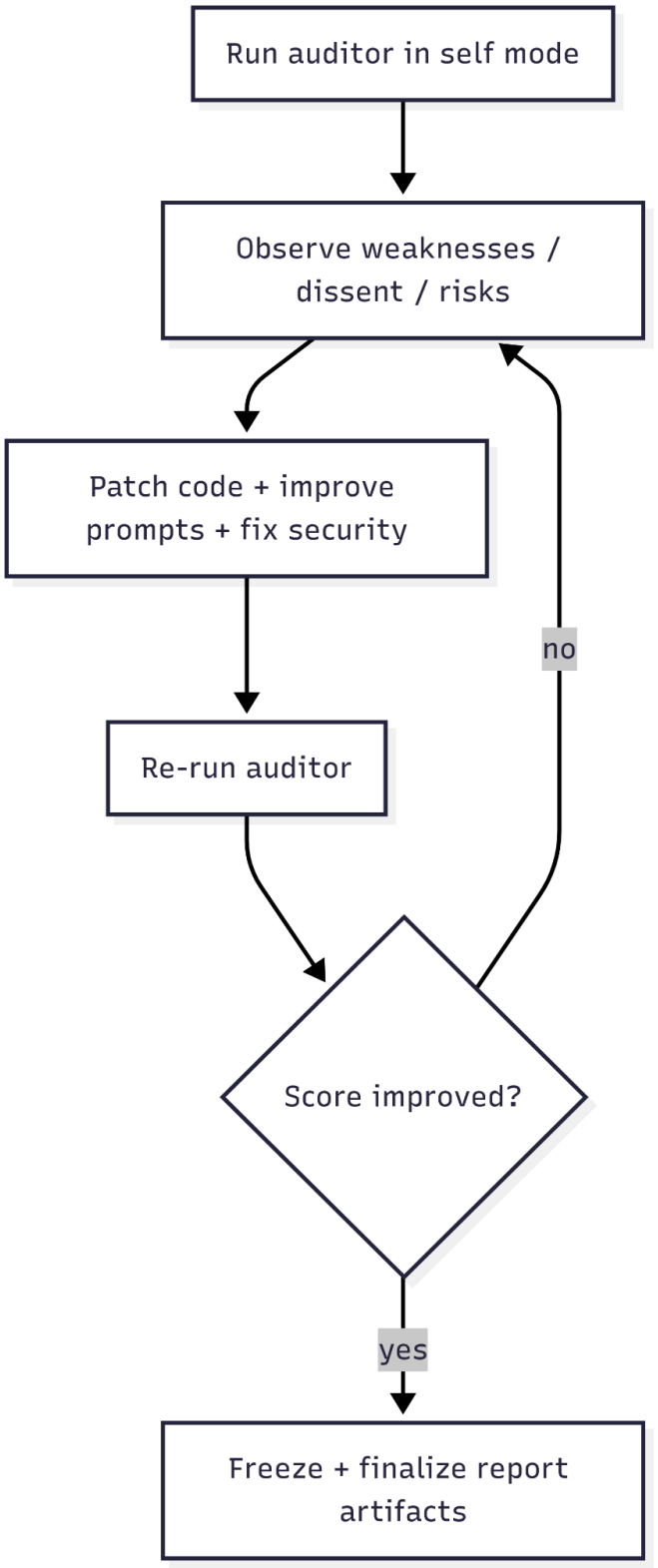


Diagram D — MinMax feedback loop (iteration cycle)



3) Self-Audit Criterion Breakdown (How I measured myself and what the results mean)

This section explains how the self-audit is computed and what “grading input” actually is.

3.1 What is the “input” to scoring?

The scoring is based on:

1. **Rubric dimensions** loaded from `rubric.json`
2. **Evidence objects** produced by detectives
3. **Judicial opinions** (score + argument + cited_evidence) produced per criterion
4. **Chief Justice deterministic synthesis logic** aggregating judge scores

So the judges are not grading randomly. They are grading *based on evidence* you collected.

A single criterion is graded like this:

- Evidence exists (or missing)
- Judges see evidence summary + criterion instruction
- Judges output structured score 1..5
- Chief Justice averages scores (with variance logic + dissent)
- Overall score computed from criterion scores

3.2 How the rubric is used (and why you shouldn’t “change it”)

Your `rubric.json` is the official rubric from the challenge doc (it uses `dimensions`, `rubric_metadata`, and `synthesis_rules`). That file defines:

- IDs for each dimension (e.g., `forensic_accuracy_code`, `langgraph_architecture`)
- instructions for what to verify
- judge role logic hints
- synthesis rules like security override and dissent requirement

You **should not rewrite this rubric** unless the challenge told you to. Your job is to ensure:

- your detectors collect evidence that maps cleanly to those dimension IDs
- your judges evaluate based on the evidence
- your Chief Justice processes those opinions deterministically

3.3 Breakdown of each rubric dimension (what my system checks)

Below is how your system is aligned to the rubric dimensions you printed (6 dimensions):

(1) `git_forensic_analysis`

- Goal: assess development progression and commit history quality.
- Evidence source: `extract_git_history(repo_path)` (repo_tools)
- What should be improved: add explicit evidence in `repo_investigator` that summarizes commit progression (e.g., number of commits, meaningful messages, chronological evolution). Right now your score is around 3 because the evidence isn't strongly presented to judges.

(2) `security_sandboxing`

- Goal: verify safe cloning & avoid unsafe execution.
- Evidence source: `clone_repo_sandboxed` (subprocess list, no shell=True), unsafe scan.
- Current status: you fixed unsafe patterns; `unsafe_files: []` is good.
- Improvement: add mention of restricted directories + safe subprocess patterns as explicit evidence.

(3) `forensic_accuracy_code`

- Goal: verify architecture exists in code (StateGraph, Typed State reducers).
- Evidence source: `verify_graph_forensics()` which checks:
    - `parallel=True` (fan-out)
    - `typed_state=True` (TypedDict + operator.ior)

- Current status: strong evidence exists (score ~4)

(4) `forensic_accuracy_docs`

- Goal: verify the PDF's claims match repo reality; detect hallucinations.
- Evidence source: PDF semantic chunks + targeted search for "Dialectical Synthesis", "Metacognition", etc.
- Weakness you saw: score ~3 because your current interim PDF didn't have enough deep detail or explicit cross-references.

(5) `judicial_nuance`

- Goal: judges must have distinct prompts + structured output.
- What happened in your run: Prosecutor sometimes scored 1 claiming "no proof of distinct prompts". That means your evidence did not include **proof that judges prompts differ**, and/or there is not a specific evidence item pointing at `src/nodes/judges.py` prompt text.

Fix: In repo_detective evidence, add a new evidence item:

- location: `src/nodes/judges.py`
- content: short excerpts showing persona strings differ for Prosecutor/Defense/TechLead
- found=True
  Then Prosecutor cannot claim "no proof".

(6) `langgraph_architecture`

- Goal: verify fan-out/fan-in across detectives and judges; reducers; error handling edges.
- You already have the fan-out/fan-in. To reach 5:
  - add conditional edges or explicit failure handling (optional)
  - include evidence demonstrating fan-in aggregator nodes exist
  - include reducers evidence

4) MinMax Feedback Loop Reflection (What I changed after feedback and why)

This project naturally forced a “MinMax” iteration: I ran the system, observed its strictest critic (Prosecutor), then improved the weakest areas.

Key iteration improvements:

1. **Security cleanup**
    - Early versions were flagged because “unsafe system execution” patterns appeared in evidence.
    - I refactored repo cloning to use safe subprocess calls (no `shell=True`).
    - I improved the unsafe scanner to skip `.venv`, caches, and self-files to avoid false positives.
      Result: `unsafe_files=[]` and security risk reduced.
2. **Evidence quality improvements**
    - I changed evidence to be shorter but more explicit (clipped snippets, clear locations).
    - I deduplicated PDF findings to prevent multiple chunks causing “inconsistent evidence” complaints.
3. **Structured output enforcement**
    - Judges now use `with_structured_output(JudicialOpinion)` so outputs conform to schema.
    - This removes messy free-text judge outputs and improves reproducibility.
4. **Chief Justice fairness**
    - Instead of hard-capping overall score to 2 when security is flagged, I applied a penalty approach.
    - This avoids a “destroy everything” outcome and keeps scoring fair.

The loop taught me something important: **even good code can score low if evidence is not presented clearly**. The auditor is only as strong as its evidence traceability.

---

5) Remediation Plan (Exactly what I'll improve next, in order)

This remediation plan is focused on raising the system's self-audit scores and also making the auditor useful for peer grading.

Priority 1 — Fix judicial_nuance score (make judges prompts provable)

Add explicit repo evidence:

- Read `src/nodes/judges.py`
- Extract 2–3 short lines from each persona prompt
- Create evidence objects proving prompt differences
  This directly blocks the Prosecutor's "persona collusion" accusation.

Priority 2 — Improve git_forensic_analysis evidence

Add a dedicated evidence object summarizing:

- total commits
- first and last commit messages
- progression pattern (setup → detection → judging → synthesis)
  That will push the score from ~3 to 4+.

Priority 3 — Strengthen forensic_accuracy_docs with cross-references

In the PDF, add:

- a section that explicitly says "Claim: parallel judges implemented in graph.py lines X–Y"
- include screenshots or code snippets and references
  This helps the Prosecutor validate "no hallucination".

Priority 4 — Extend langgraph_architecture robustness

Optional improvements for maximum score:

- Add conditional edges or failure handling (e.g., if repo cloning fails, bypass some nodes).

- Add explicit state flags like `repo_ok`, `pdf_ok`.
  This makes the orchestration “rigor” stronger.

Priority 5 — Improve output artifacts

Ensure every run creates:

- JSON final report

- Markdown audit report

- saved folder under `audit/report_*`

- include LangSmith trace screenshot/link if required by submission