Automaton Auditor

Week 2 Interim Architecture Report

Submission Type	Interim — Detective Layer
Phase Completed	Phase 1 (State) + Phase 2 (Tools + Detectives)
Phase Planned	Phase 3 (Judges) + Phase 4 (Chief Justice) — due Saturday
LLM Provider	Anthropic Claude (claude-opus-4-6)
Orchestration Framework	LangGraph StateGraph

1. Executive Summary

The Automaton Auditor is a hierarchical multi-agent swarm built with LangGraph, designed to forensically evaluate Week 2 submissions. The system implements the "Digital Courtroom" architecture: Detective agents gather hard evidence from code and documents, Judge agents debate that evidence from adversarial personas, and a Chief Justice synthesises a final verdict using deterministic rules — not LLM guesswork.

This interim submission delivers the complete Detective layer: state definitions, forensic tooling, and the parallel fan-out/fan-in graph wiring for RepoInvestigator and DocAnalyst. The Judicial layer will be completed for the Saturday final submission.

2. Architecture Decisions

2.1 Why Pydantic Over Plain Dicts

The assignment warned against "Dict Soups" — and for good reason. Plain Python dicts fail silently when a key is missing or a value has the wrong type. In a multi-agent system where a Prosecutor might receive malformed evidence and quietly produce a corrupted score, silent failures are catastrophic.

Every data structure in this system — Evidence, JudicialOpinion, CriterionResult, AuditReport — inherits from Pydantic's BaseModel. This provides: automatic type validation on construction, clear error messages when a Detective returns malformed data, and free JSON serialisation for LangSmith tracing. AgentState itself uses TypedDict (required by LangGraph) with Pydantic models as values.

2.2 State Reducers: operator.ior and operator.add

LangGraph runs parallel nodes in separate threads. Without reducers, two Detectives writing to the same state key would race — the last write wins and the other is silently lost. The Annotated[T, reducer] pattern tells LangGraph how to merge concurrent writes:

evidences uses operator.ior (dict union). RepoInvestigator writes {"git_forensic_analysis": [Evidence(...)]} and DocAnalyst writes {"theoretical_depth": [Evidence(...)]}. The ior reducer merges these dicts — neither overwrites the other. opinions uses operator.add (list concatenation). Each of the three parallel Judges appends their JudicialOpinion; the add reducer builds the full list rather than overwriting it.

2.3 AST Parsing vs. Regex

The forensic instruction explicitly says "Do not rely on Regex. It is brittle." Regex breaks when someone writes StateGraph(with unusual spacing, or splits an instantiation across lines. Python's built-in ast module parses code into a proper syntax tree, making checks like "does this file contain a class that inherits from BaseModel" structurally sound and formatting-independent.

The analyze_graph_structure() function in src/tools/repo_tools.py walks the AST of graph.py and state.py to detect: StateGraph instantiation, add_edge/add_conditional_edges calls (fan-out detection), BaseModel and TypedDict inheritance, and operator.add/ior reducer usage — all without a single regex.

2.4 Sandboxed Git Cloning

The rubric's "Security Negligence" ruling explicitly penalises os.system('git clone') that drops code into the live working directory. The clone_repo() function in src/tools/repo_tools.py uses tempfile.TemporaryDirectory() as a context manager. The cloned code lives in an OS-managed temp directory and is deleted automatically when the context manager exits, even if an exception occurs.

subprocess.run() is used instead of os.system() because it captures stdout/stderr separately, checks return codes, supports timeouts (120 seconds to prevent hanging on slow repos), and does not spawn a full shell (preventing injection if the URL somehow contains shell metacharacters). A URL allowlist (github.com, gitlab.com only) provides an additional layer of input sanitisation.

3. StateGraph Flow — Interim Submission

The diagram below shows the current detective fan-out/fan-in pattern. Parallel execution is represented by the split after context_builder — both RepoInvestigator and DocAnalyst run concurrently, and their results are merged by the EvidenceAggregator via the operator.ior state reducer before LangGraph invokes the next node.

Node	Role	Parallel?
START	Graph entry point	—
context_builder	Loads rubric.json into state	No
repo_investigator	Clones repo, runs AST + git forensics	YES (parallel with doc_analyst)
doc_analyst	Ingests PDF, checks concept depth	YES (parallel with repo_investigator)

evidence_aggregator	Fan-in sync — waits for both detectives	No
[Judges — Final Sub]	Prosecutor || Defense || TechLead	YES (parallel, 3-way)
[ChiefJustice — Final Sub]	Deterministic synthesis + Markdown report	No
END	Graph exit point	—

Table 1: StateGraph nodes — grey rows are planned for final submission.

4. Fan-In / Fan-Out: How Parallelism Works

Fan-Out is the pattern where a single node triggers multiple parallel branches. In LangGraph this is implemented by calling builder.add_edge(source, nodeA) and builder.add_edge(source, nodeB) from the same source node — LangGraph spawns both nodeA and nodeB as concurrent tasks.

Fan-In is the synchronisation pattern where parallel branches must all complete before the graph continues. In LangGraph, when multiple nodes all add edges to the same destination node, LangGraph automatically waits for all of them to finish before invoking the destination. The state reducers (operator.ior, operator.add) handle the merge so that no branch's output is lost.

The detective layer demonstrates Fan-Out at context_builder (both detectives launch simultaneously) and Fan-In at evidence_aggregator (neither the Judges nor the ChiefJustice can run until both detectives have deposited their evidence). The final submission will add a second Fan-Out/Fan-In pair for the three Judges.

5. Known Gaps and Plan for Final Submission

Gap	Plan for Saturday
VisionInspector not implemented	Add src/nodes/detectives.py VisionInspector using Claude's vision API to classify diagram flow
Judges not wired (src/nodes/judges.py missing)	Implement Prosecutor, Defense, TechLead with .with_structured_output() bound to JudicialOpi
ChiefJustice not implemented	Implement deterministic Python conflict resolution rules in src/nodes/justice.py
Graph ends at EvidenceAggregator	Wire full graph: EvidenceAggregator -> [3 Judges parallel] -> ChiefJustice -> Markdown report
No Markdown report output	ChiefJustice will serialise AuditReport to a .md file in audit/
LangSmith not enabled yet	Add LANGCHAIN_TRACING_V2=true and verify traces appear for full end-to-end run

Table 2: Known gaps with concrete remediation plan.

6. Dialectical Synthesis — The Core Reasoning Model

Dialectical Synthesis is the philosophical process of resolving opposing viewpoints (Thesis vs. Antithesis) into a higher-order Synthesis. In this system, the Thesis is the Defense Attorney's argument ("This code shows deep architectural understanding"), and the Antithesis is the Prosecutor's argument ("This graph is a linear spaghetti script that violates the parallelism requirement"). The Synthesis is not an average of the two — it is the Chief Justice applying deterministic rules to decide which argument is grounded in forensic fact and which is opinion.

The Rule of Evidence implements fact supremacy: if the Defense claims "deep metacognition" but the RepoInvestigator found no PDF report or no parallelism in the graph, the Detective's forensic finding overrules the Judge's interpretation. This prevents the system from grading on vibes rather than evidence — exactly the failure mode the assignment is designed to resist.

7. Metacognition in the Auditor Architecture

Metacognition — thinking about thinking — is embedded in the architecture at two levels. At the object level, the three Judges each evaluate the same evidence independently, producing different assessments of the same facts. At the meta level, the Chief Justice evaluates the quality of the Judges' reasoning: it checks whether the Defense's claims are supported by Detective evidence, and whether the Prosecutor's charges are confirmed security violations or speculative criticism.

This means the system is not just grading code — it is grading its own grading process. When score variance exceeds 2 (e.g., Prosecutor says 1, Defense says 5), the variance_re_evaluation rule triggers a second pass that forces the Chief Justice to explicitly cite which Detective evidence supports which Judge's position before rendering a final score. This is the system thinking critically about the quality of its own judicial process.

8. Delivered File Structure (Interim)

The following files are committed to the repository for the interim submission:

File	Purpose
`src/state.py`	Evidence, JudicialOpinion, AuditReport, AgentState with reducers
`src/tools/repo_tools.py`	Sandboxed git clone, git log, AST graph analysis
`src/tools/doc_tools.py`	PDF ingestion, RAG-lite query, concept depth analysis
`src/nodes/detectives.py`	RepoInvestigator, DocAnalyst, EvidenceAggregator nodes
`src/graph.py`	StateGraph: detective fan-out/fan-in, run entry point
`rubric.json`	Machine-readable rubric — the agent's Constitution
`pyproject.toml`	uv-managed dependencies
`.env.example`	Required API keys (no secrets)
`README.md`	Setup and run instructions
`reports/interim_report.pdf`	This document