Automaton Auditor — Architecture & Progress Report

Purpose: Document architecture decisions, current implementation status, known gaps, and a concrete forward plan for the judicial layer and synthesis engine.

---

1. Architecture Decision Rationale

This section justifies core technical choices with trade-off analysis, alternatives considered, and the specific failure modes each decision prevents.

1.1 Pydantic / TypedDict Over Plain Dicts for State

Decision: All graph state and LLM-produced data use Pydantic `BaseModel` for `Evidence`, `JudicialOpinion`, `CriterionResult`, and `AuditReport`, and TypedDict with `Annotated` reducers for `AgentState` (e.g. `Annotated[dict[str, list[Evidence]], operator.ior]`).

Why this prevents failures:

- Parallel agents cannot corrupt a shared state with untyped dicts. With plain dicts, a Detective could return `{"evidences": "broken"}` or `{"evidences": {1: "not a list"}}` and the reducer would merge it; downstream Judges or Chief Justice would then see invalid structure and crash or produce nonsense. Pydantic validates at the boundary: only `Evidence` instances (or dicts that validate to `Evidence`) are allowed in `evidences` values, and `JudicialOpinion` enforces `score` in 1–5 and `judge` in `Literal["Prosecutor","Defense","TechLead"]`. So a malformed LLM response is caught before it enters state.
- Reducer semantics are type-safe. `operator.ior` and `operator.add` are defined over the annotated types; LangGraph merges updates correctly. Plain dicts would allow accidentally overwriting entire keys (e.g. one Detective returning `{"evidences": {...}}` that overwrites another's key) if the reducer were not applied consistently; the type contract makes "partial update" explicit.
- IDE and refactoring: Typed state enables safe renames and field changes; plain dicts do not.

Alternatives considered and rejected:

- TypedDict only (no Pydantic for Evidence/Opinion): TypedDict does not validate at runtime. An LLM could return `{"score": 10}` and it would be accepted until something downstream failed. We need runtime validation for any LLM-produced structure.
- Plain dicts: Rejected for the reasons above; no validation, no reducer guarantee, and high risk of silent corruption when multiple nodes write concurrently.

Costs: Pydantic adds a dependency and requires schema updates when contracts change; serialization/deserialization (e.g. after checkpointing) must round-trip through these models. We accepted this for correctness and debuggability.

1.2 AST Parsing for Graph Structure (Not Regex)

Decision: Repo forensic tool `analyze_graph_structure(repo_path)` uses Python's `ast` module to parse `src/graph.py` (or `graph.py`), walking the AST to detect `StateGraph`, `add_edge`, `add_node`, `add_conditional_edges`, and reducer usage. We do not rely on regex for structure.

Why this prevents failures:

- Regex breaks on multiline and nested structures. A regex like `add_edge\s*\(\s*["']](\w+)["']` can miss edges where the first argument is split across lines or is a variable. Multiline class definitions or nested calls (e.g. `add_edge(foo(), bar())`) are brittle. AST parses the real structure: we only care about call names and constant arguments, so we use `ast.Call`, `ast.Attribute`, and `ast.Constant` to get exact node names and edge endpoints. That way we correctly detect fan-out (multiple edges from the same source) and fan-in (multiple edges into the same target) regardless of formatting.
- Syntax errors are handled explicitly. If the target file has a syntax error, `ast.parse()` raises; we catch it and return a structured result with `error` set and `file_found: True`, so the Detective can report "graph file present but unparseable" instead of misreporting "no parallelism" due to a failed regex.
- Nested and dynamic structure: Logic like "reducer usage" is detected by walking the AST for `operator.ior` / `operator.add` in the state definition and in function bodies. Regex would be unreliable for that (e.g. inside nested blocks or with different formatting).

Alternatives considered and rejected:

- Regex-only: Rejected for the reasons above; acceptable for trivial one-line patterns only.
- Tree-sitter: Mentioned in _meta as an option; we chose stdlib `ast` for zero extra dependencies and sufficient accuracy for Python-only repos. Tree-sitter would be preferable for multi-language codebases.

Costs: AST is Python-specific; we only analyze Python graph files. For a multi-language auditor, a tree-sitter–based or language-specific pipeline would be needed.

1.3 Sandboxing Strategy for Cloning Unknown Repos

Decision: All git operations run in a temporary directory created with `tempfile.mkdtemp(prefix="automaton_auditor_clone_")`. We use `subprocess.run()` with an explicit argument list (`["git", "clone", "--depth", "1", url, tmp_dir]`), with `timeout=120`, and we never pass user-controlled strings to `os.system()` or to a shell.

Why this prevents failures:

- No code execution from URL or path. If we used `os.system(f"git clone {repo_url}")`, a malicious URL like `https://evil.com; rm -rf /` could lead to shell injection. Subprocess with a list of arguments does not invoke a shell; the URL is passed as a single argument to `git clone`, so it is not interpreted as code.
- Isolation: The clone lives in a temp dir that is not the process cwd. Even if the repo contained a malicious hook or script, the working directory of the auditor is not the repo; we only read from the cloned path. Cleanup is the caller's responsibility (or process exit); we do not execute any repo code.
- Predictable failure: Clone failures (invalid URL, auth, network) raise `RepoCloneError` with a clear message; the graph can route to an error handler or inject placeholder evidence (SRS FR-19, A2) instead of failing in an undefined way.

Alternatives considered and rejected:

- Clone into cwd or a fixed directory: Risk of overwriting existing data and no isolation; rejected.

- Docker or VM per clone: Strong isolation but heavy; we deferred to “temp dir + subprocess” for the current scope. For higher assurance, a containerized clone step could be added later.

Costs: Temp dirs consume disk; we use `--depth 1` to limit size. No network isolation (we rely on git’s behavior).

1.4 RAG-Lite for PDF Ingestion (Not Full-Doc Dump)

Decision: PDF ingestion uses chunked extraction (pypdf page text → split into overlapping chunks, e.g. 1500 chars with 100 overlap). We expose a queryable store (`DocStore`) with a simple keyword-overlap search; Detective nodes call `query_doc(store, question)` so only relevant chunks are passed to the LLM or into evidence, not the full document.

Why this prevents failures:

- Context overflow and noise: Dumping the full PDF into a single string and passing it to an LLM would exceed context limits for large reports and dilute relevant parts. Chunking keeps each unit bounded; retrieval selects only a few chunks per question, so we stay within token limits and improve relevance.
- Structured evidence: Evidence can cite “excerpt from chunk X” rather than “the whole report,” which aligns with the rubric’s requirement for factual, citable findings.

Alternatives considered and rejected:

- Full-doc single string: Rejected for the reasons above.
- Full embedding RAG: We use a simple keyword-overlap search (no vector DB) to avoid extra dependencies and to keep the “RAG-lite” contract (SRS FR-8). For higher accuracy, embeddings + vector stores could be added later.

Costs: Simple keyword search may miss paraphrased or conceptual matches; that’s an accepted trade-off for the current implementation.

1.5 LLM Provider and Structured Output for Judges

Decision: Judge nodes use LangChain with OpenAI (`ChatOpenAI`, e.g. `gpt-4o-mini`) and `.with_structured_output(JudicialOpinion)` so every Judge response is

parsed and validated against the Pydantic `JudicialOpinion` schema. We use distinct system prompts per persona (Prosecutor, Defense, Tech Lead) and retry up to three times on parse failure; we do not append invalid output to state.

Why this prevents failures:

- Non-deterministic output: Without structured output, the LLM could return free text that does not match the expected fields; downstream synthesis would break. Binding to `JudicialOpinion` ensures `judge`, `criterion_id`, `score` (1–5), `argument`, and `cited_evidence` are always present and validated.
- Persona drift: We enforce the `judge` field in code after the call (overwriting the model's response with the intended persona) so that even if the model returns the wrong role name, the state remains consistent.

Alternatives considered: Other providers (Anthropic, Gemini) could be wired via LangChain; we standardized on OpenAI for the current implementation. Structured output could be implemented with `bind_tools()` and the same schema; we use `with_structured_output` for simplicity.

---

2. Known Gaps and Forward Plan

Honest self-assessment of what is not yet fully built and a concrete, actionable plan for the judicial layer and synthesis engine.

2.1 Known Gaps

- Conditional edges not wired: The graph is linear after each layer (no `add_conditional_edges`). SRS FR-19 and A2 require branching for "Evidence Missing" or "Node Failure" (e.g. clone failure, missing PDF). Today, failures are handled inside nodes (exceptions, placeholder evidence in EvidenceAggregator); there is no explicit graph branch to an "error" or "fallback" node. So the flow does not yet show conditional edges in the diagram, and an evaluator scanning the graph code would not see `add_conditional_edges`.
- Judge parallelism is three-way, not per-criterion parallel in the graph: Prosecutor, Defense, and Tech Lead run as three parallel nodes (fan-out from EvidenceAggregator, fan-in at judge_collector). Each node, however, loops over all rubric dimensions sequentially and produces one opinion per dimension. So we have "three judges in parallel" but not "for each criterion, three judges in parallel" as separate graph nodes. The outcome (three opinions per criterion) is

correct; the graph topology could be extended to a subgraph per dimension if we wanted to maximize per-criterion parallelism and make it visible in the diagram.

- Persona convergence risk: LLMs can still produce similar scores/arguments despite different system prompts. We have not added few-shot examples or stronger “adversarial vs forgiving” examples in the prompts; monitoring variance and dissent rates would help detect convergence.
- Parse failure handling: On repeated structured-output parse failure we skip that dimension (no opinion appended). The graph does not branch to a “retry” or “human review” node; the plan below includes making this explicit (e.g. placeholder opinion or conditional edge).
- Synthesis rules and rubric alignment: Chief Justice applies hardcoded rules (security_override, fact_supremacy, functionality_weight, dissent_requirement, variance_re_evaluation). The rubric’s `synthesis_rules` are loaded but the exact text is not yet used to drive behavior (logic is in code). Aligning rule labels with rubric text would improve traceability.

2.2 Concrete Plan: Judicial Layer

1. Persona differentiation (prompts)
    - Action: Add 1–2 few-shot examples per persona (e.g. “For this evidence, Prosecutor argues 2 because …; Defense argues 4 because …”) so the model sees the desired spread.
    - Action: Load persona-specific text from rubric if `judicial_logic` or per-persona fields exist; otherwise keep current system prompts.
    - Risk: LLM may still converge; mitigate by checking score variance in tests and, if needed, increasing temperature slightly for Defense/Prosecutor.
2. Structured output enforcement
    - Action: Keep `.with_structured_output(JudicialOpinion)`; add an explicit “on third failure” path: either inject a placeholder `JudicialOpinion` (score=3, argument="Parse failure; no opinion") or emit a dedicated state flag (e.g. `opinion_parse_failures: list[str]`) so Chief Justice can report “Dimension X: Judge output invalid.”
    - Action: Add a unit test that mocks an LLM returning invalid JSON and asserts we do not append to `opinions` and either retry or record the failure.
3. Parallel execution topology
    - Option A (current): Keep three nodes (Prosecutor, Defense, Tech Lead), each iterating over dimensions. Easiest; already satisfies “three opinions per criterion.”
    - Option B (optional): Introduce a subgraph or map over dimensions: for each dimension, fan-out to three judge nodes, then fan-in. This would make “per-criterion parallel” visible in the diagram and in the AST; it

increases node count and complexity. Sequence: Implement only if rubric explicitly requires “per-criterion” parallelism in the graph structure; otherwise document current behavior as compliant.

4. Conditional edges (error paths)
    - Action: After START or after EvidenceAggregator, add `add_conditional_edges` that check for “no evidence for dimension X” or “clone/pdf failure” and route to a small “error_handler” node that injects placeholder evidence or sets a state flag, then rejoin. This satisfies SRS FR-19 and makes error handling visible in the diagram.

2.3 Concrete Plan: Synthesis Engine

1. Hardcoded rules (already implemented)
    - Action: Document in code comments the mapping: `security_override` → `_evidence_has_security_issue` + cap at 3; `fact_supremacy` → `_evidence_supports_claim` and lowering Defense when unsupported; `functionality_weight` → Tech Lead score for architecture/graph dimensions; `dissent_requirement` / `variance_re_evaluation` → variance > 2 → dissent_summary and Tech Lead re-evaluation.
    - Action: Add tests: mock opinions with variance 3, assert `dissent_summary` is present; mock evidence with “os.system”, assert final score ≤ 3.
2. Dissent and variance re-evaluation
    - Action: Keep current logic (variance > 2 → set dissent_summary, use Tech Lead as tie-breaker). Optionally: when variance > 2, include a one-line summary of each judge’s score in the dissent text (already partially done).
    - Risk: Edge cases where only two of three opinions exist (e.g. one parse failure); ensure we don’t compute variance on a single opinion and mislabel dissent.
3. Rubric-driven rule text
    - Action: Read `synthesis_rules` from rubric and pass rule *labels* into Chief Justice so logging or future UI can show “rule X applied.” No change to the deterministic logic itself unless we later move to config-driven thresholds (e.g. “security_cap” value in rubric).

2.4 Sequencing and Handoff

- Phase 1 (quick): Add conditional edges from EvidenceAggregator for “missing evidence” / “failure” and an error_handler node; update diagram.
- Phase 2: Strengthen Judge prompts (few-shot, rubric judicial_logic) and add parse-failure handling (placeholder or state flag + test).

- Phase 3: Document synthesis rules in code and add variance/dissent and security-cap tests.
- Phase 4 (optional): Per-criterion judge subgraph if required by rubric.

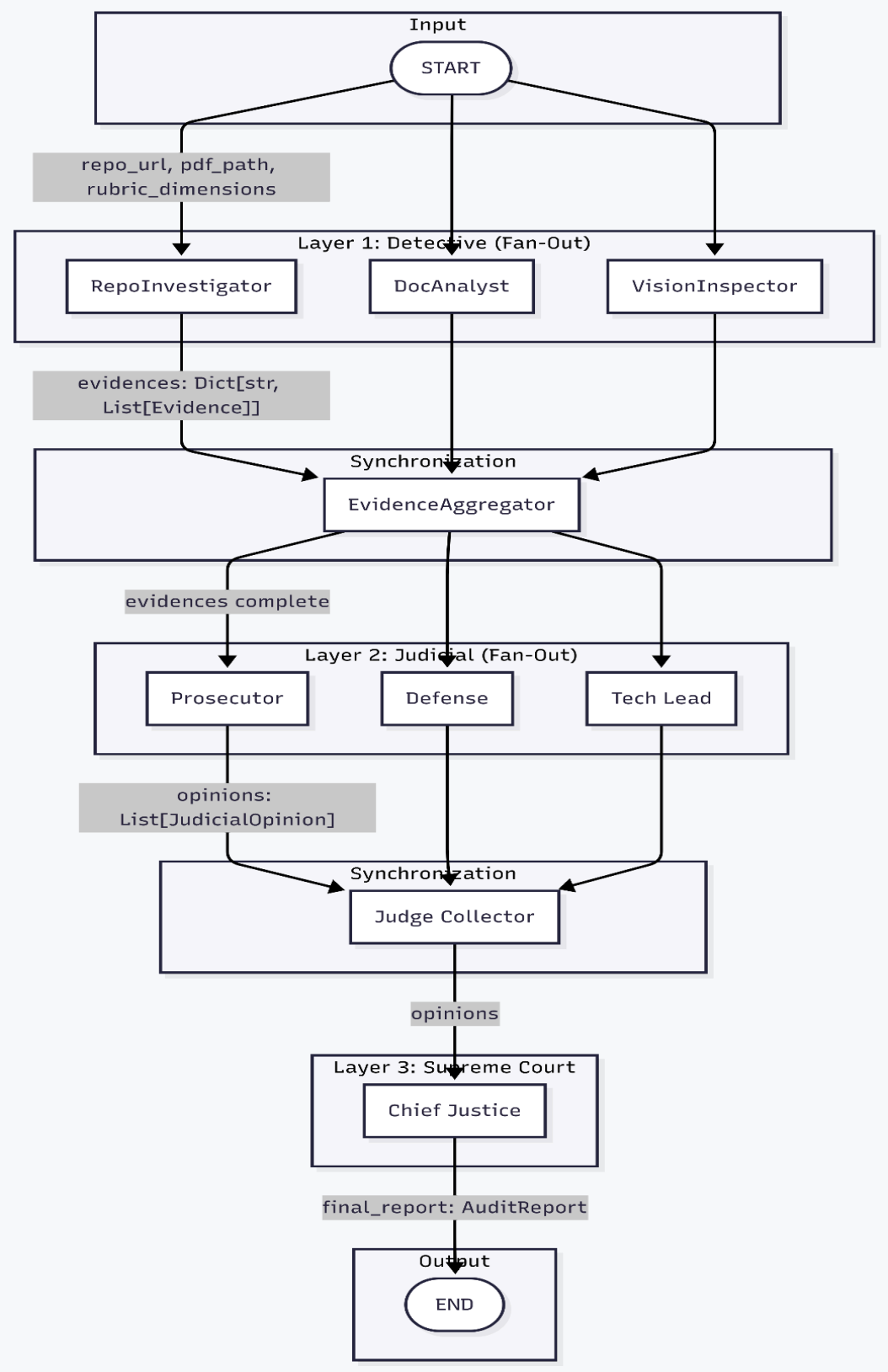
Another engineer can pick up this plan: the judicial layer is “persona prompts + structured output + optional graph expansion”; the synthesis engine is “document rules, add tests, optionally wire rubric labels.”

---

3. StateGraph Architecture Diagram

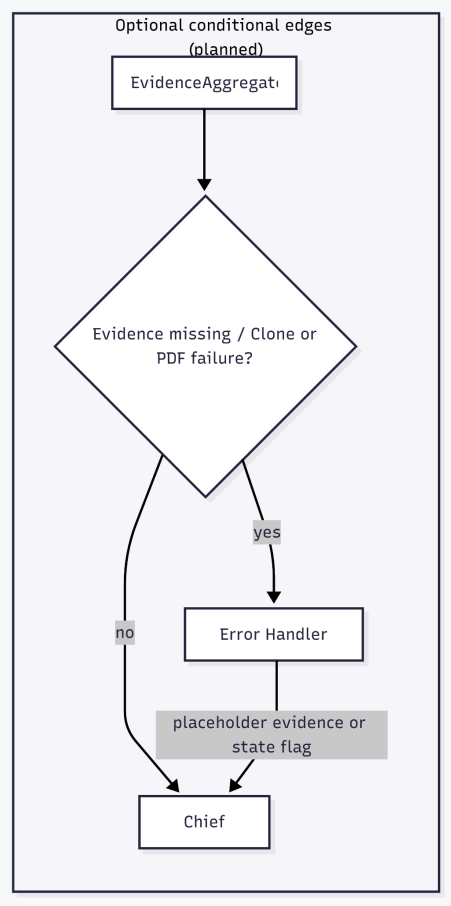
The following diagrams show the planned Digital Courtroom flow with both detective and judicial fan-out/fan-in, synchronization nodes, state types on edges, and optional error paths.

3.1 Full Flow (Detectives + Judicial + Supreme Court)


Input
START
repo_url, pdf_path, rubric_dimensions
Layer 1: Detective (Fan-Out)
RepoInvestigator
DocAnalyst
VisionInspector
evidences: Dict[str, List[Evidence]]
Synchronization
EvidenceAggregator
evidences complete
Layer 2: Judicial (Fan-Out)
Prosecutor
Defense
Tech Lead
opinions: List[JudicialOpinion]
Synchronization
Judge Collector
opinions
Layer 3: Supreme Court
Chief Justice
final_report: AuditReport
Output
END

3.2 Conditional / Error Paths (Planned)

The following is not yet implemented but is in the forward plan (SRS FR-19, A2):



- From START or after Detectives: If clone fails or PDF is missing, conditional edge routes to an error handler that injects placeholder evidence (or sets a flag) so the graph can continue to EvidenceAggregator and then Judges, or terminate cleanly.
- After EvidenceAggregator: If a dimension has no evidence, we already inject a placeholder in the aggregator; an explicit conditional could route to a “partial evidence” path for logging or reporting.

3.3 State Types on Edges (Summary)

Edge	Data / state
START → Detectives	`repo_url`, `pdf_path`, `rubric_dimensions`

Detectives → EvidenceAggregator	Partial `evidences` (reducer: `operator.ior`)
EvidenceAggregator → Judges	Full `evidences` (and rubric_dimensions)
Judges → Judge Collector	Partial `opinions` (reducer: `operator.add`)
Judge Collector → Chief Justice	Full `opinions`
Chief Justice → END	`final_report` (`AuditReport`)

---

4. Summary

- Architecture decisions are justified with trade-offs and failure-mode reasoning: Pydantic/TypedDict prevent parallel state corruption and enable validation; AST avoids regex brittleness on structure; sandboxing (temp dir + subprocess, no `os.system`) prevents injection; RAG-lite avoids context overflow; structured output for Judges ensures valid opinions.
- Gaps include: no conditional edges in the graph yet, Judge parallelism is three-way (not per-criterion in the graph), persona convergence risk, and parse-failure handling that skips dimensions without a visible error path. The forward plan is sequenced: (1) conditional edges + error handler, (2) stronger Judge prompts and parse-failure handling, (3) synthesis documentation and tests, (4) optional per-criterion judge subgraph.
- StateGraph diagrams show both detective and judicial fan-out/fan-in, EvidenceAggregator and Judge Collector as synchronization points, state types on edges, and a planned conditional/error path for evidence missing or node failure.