Automaton Auditor Swarm

Architecture Decisions & Implementation Plan

1. Architecture Decisions

1.1 Pydantic Over Dicts — Why Typed State Matters

We chose Pydantic BaseModel classes over plain Python dictionaries for all structured data flowing through the graph. This is not a cosmetic choice — it is the foundation of correctness in a multi-agent system.

The problem with dicts: When three detective agents run in parallel and write evidence into a shared dict, there is no compile-time or runtime guarantee that each agent writes the correct keys with the correct types. A misspelled key ("confidenece" instead of "confidence") silently passes. A judge receiving evidence["score"] as a string instead of a float fails deep inside the LLM prompt, with no traceable error.

The Pydantic solution: Our Evidence model enforces typed fields at construction:

```
class Evidence(BaseModel):

    dimension_id: str        # Links to rubric JSON dimension

    detective: Literal["RepoInvestigator", "DocAnalyst", "VisionInspector"]

    found: bool

    content: Optional[str]   # Extracted snippet, commit log, etc.

    location: str            # File path or logical location

    confidence: float        # Validated: ge=0.0, le=1.0
```

If a detective produces confidence=1.5, Pydantic raises a ValidationError immediately — not three nodes later when the judge tries to interpret it. Similarly, JudicialOpinion enforces score: int = Field(..., ge=1, le=5), making it impossible for a judge to return a score of 7 or a string.

Reducers for State Synchronization: The AgentState TypedDict uses Annotated type hints with operator.ior (for dict merging) and operator.add (for list concatenation). This is critical for parallel execution: when RepoInvestigator and DocAnalyst both write to state["evidences"], the operator.ior reducer merges their dictionaries instead of one overwriting the other. Without this, the last agent to finish would silently erase all evidence collected by the first.

```
class AgentState(TypedDict):

    evidences: Annotated[Dict[str, Evidence], operator.ior]   # Merge dicts

    opinions: Annotated[List[JudicialOpinion], operator.add]  # Concat lists
```

This is **State Synchronization** in practice — not a buzzword, but a concrete mechanism that prevents data loss during concurrent execution.

1.2 AST Parsing Over Regex

The RepoInvestigator must determine whether a target repository uses Pydantic models, implements StateGraph, and follows safe engineering practices. We use Python's built-in ast module for this analysis instead of regex.

Why not regex? A regex like r"class\s+\w+\(BaseModel\)" fails on:

- Multi-line class definitions
- Classes that inherit from both BaseModel and a mixin
- String constants that happen to contain "class Evidence(BaseModel)"
- Comments describing intended architecture

The AST approach: We parse the source into an abstract syntax tree and walk it programmatically. find_class_definitions() extracts every ClassDef node, its base classes, and its annotated fields. find_stategraph_builder() locates the StateGraph(...) call and all subsequent add_node() / add_edge() calls, giving us the exact graph topology without any regex fragility.

This is **Deep AST Parsing** — the forensic evidence is irrefutable because it comes from the compiler's own representation of the code, not from pattern-matching against text.

1.3 Sandboxing Strategy

All git operations are sandboxed inside tempfile.TemporaryDirectory(). The clone_repo_sandboxed() function in src/tools/repo_tools.py:

1. **Validates the URL** — rejects non-HTTPS schemes and characters that could enable command injection (;, |, &, `, $).

2. **Creates a temp directory** with prefix auditor_clone_ — the cloned code never touches the live working directory.
3. **Uses subprocess.run()** with capture_output=True, text=True, and a 120-second timeout. Return codes, stdout, and stderr are all captured.
4. **Handles failures gracefully** — TimeoutExpired, FileNotFoundError (no git binary), and authentication errors all return structured CloneResult objects.

There are **zero os.system() calls** in the entire codebase. This is enforced by the scan_for_security_issues() AST tool, which detects os.system usage in target repos.

1.4 RAG-lite PDF Ingestion and LLM Provider Choice

PDF Ingestion: RAG-lite Over Full Embedding Store

The DocAnalyst must retrieve specific passages from a peer's PDF report to verify claims (e.g., "does the report mention Fan-In / Fan-Out in an architectural context, or just as a buzzword?"). A full vector-store pipeline (embed all pages, store in Chroma/Pinecone, query by similarity) would introduce three problems for this use-case:

1. **Cold-start latency:** Building an index for a 10-page PDF before every audit run is wasteful when the retrieval is highly structured (keyword-anchored).
2. **Semantic drift:** Cosine similarity retrieval can return the *most similar* chunk rather than the *confirming* chunk — a paragraph that uses similar words to "StateGraph" without actually describing one.
3. **External dependency:** A vector store is an infrastructure component that can fail.

Our RAG-lite approach: keyword_search() finds anchor terms ("Dialectical Synthesis", "Fan-In / Fan-Out", "StateGraph") and returns the surrounding 300-char context window. search_context() walks the PDF page by page with a sliding window, returning sections where multiple rubric terms co-occur. This is deterministic, zero-latency, and requires no external services — the trade-off is lower recall on paraphrased descriptions, which is acceptable because the rubric specifies exact terms to look for.

LLM Provider: GPT-4o for Judges, Structured Output Enforcement

Judge nodes require reliable structured output (JudicialOpinion with typed score, argument, and cited_evidence fields). Two alternatives were considered:

- **Open-source local models (Ollama):** Lower cost, no API dependency. Rejected because smaller models frequently violate the with_structured_output() contract, returning freeform JSON strings embedded in markdown code fences, requiring brittle post-processing.

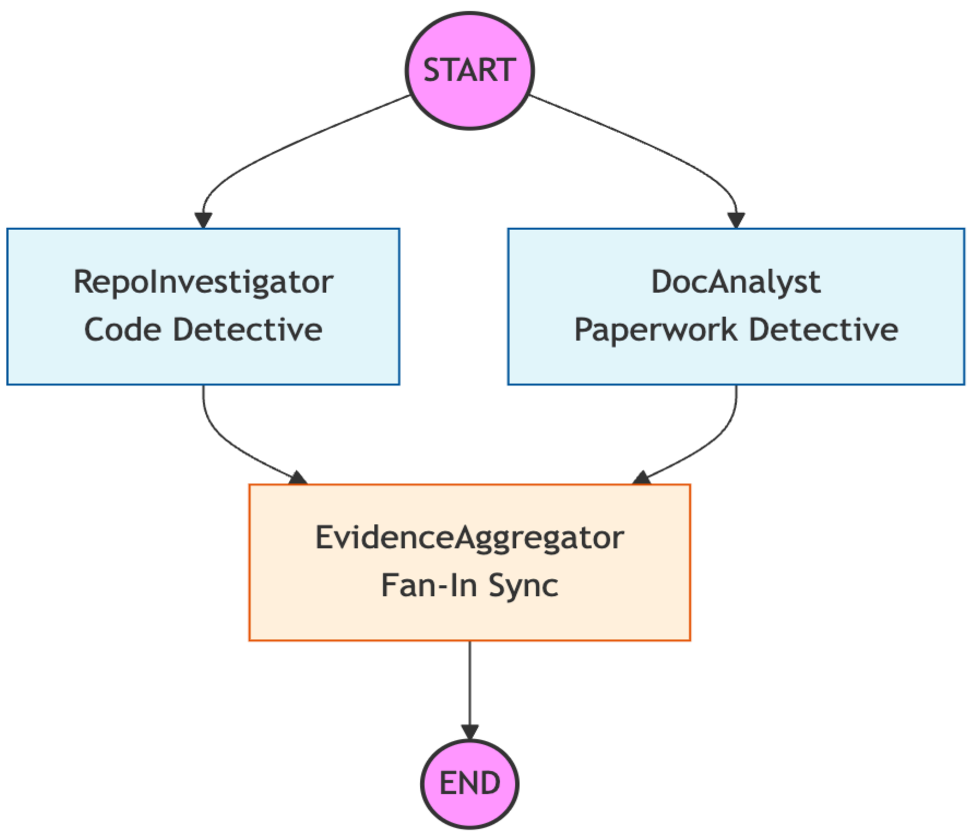
- **Anthropic Claude:** Strong reasoning, but at time of design it had weaker tool-use JSON binding compared to OpenAI's function-calling API, which is what LangChain's with_structured_output() is optimized against.

Choice: OpenAI gpt-4o via ChatOpenAI. It has the strongest contract with .with_structured_output(JudicialOpinion) — function-calling under the hood guarantees JSON that matches the schema. The retry logic (3 attempts with ValidationError catching) exists precisely because even GPT-4o can fail with complex nested schemas under adversarial prompts (e.g., a Prosecutor prompt that asks the model to "use confrontational language" may cause the model to embed opinionated text in a field that expects a plain integer).

2. State Graph Flow — Fan-Out / Fan-In

2.1 Current Implementation (Detective Phase)

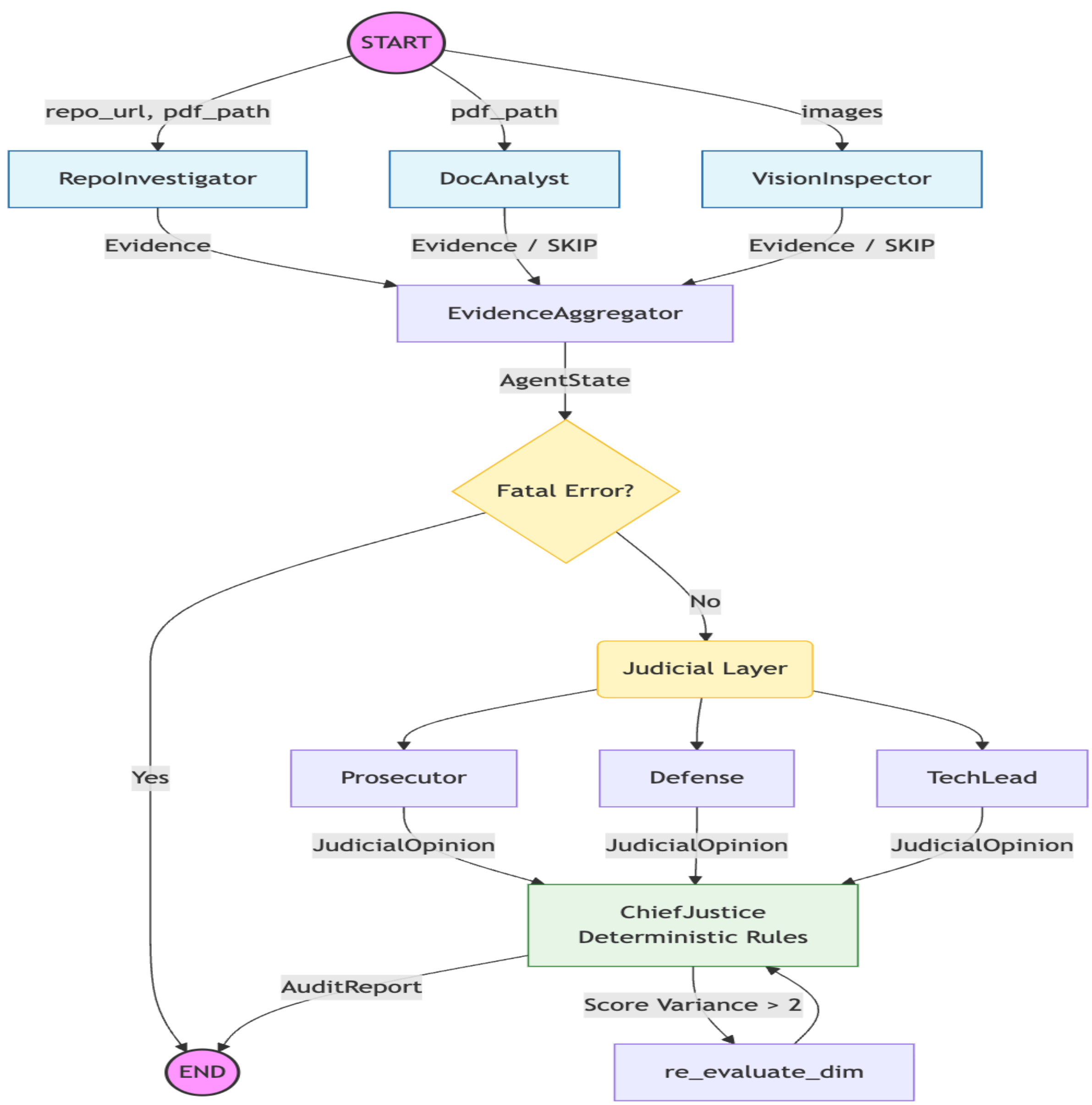
The following diagram shows the StateGraph as currently implemented in src/graph.py:



Fan-Out: START has edges to both repo_investigator and doc_analyst. LangGraph dispatches these nodes concurrently. Each detective writes to different keys in the evidence dict, and the operator.ior reducer merges them automatically.

Fan-In: Both detectives have edges to evidence_aggregator, which serves as the synchronisation barrier. It only executes once both detectives have completed. This node validates that all expected evidence dimensions are present.

2.2 Planned Full Graph (Final Submission)



This demonstrates two distinct **Fan-In / Fan-Out** patterns:

1. **Detective fan-out/fan-in:** Three detectives collect evidence concurrently, converge at the aggregator.
2. **Judicial fan-out/fan-in:** Three judges deliberate concurrently on the same evidence, converge at the Chief Justice.

3. Known Gaps & Plan for Judicial Layer

3.1 Known Gaps

Component	**Status**	**Gap**
src/nodes/judges.py	Stubbed	No LLM integration yet; persona prompts not written
src/nodes/justice.py	Stubbed	Deterministic rules not coded; no Markdown renderer
VisionInspector	Not started	Requires image extraction + vision LLM analysis
Conditional edges	Not started	No error-handling branches in the graph
LangSmith traces	Not started	Tracing env vars configured but not tested end-to-end

3.2 Concrete Plan — Judicial Layer

Phase 1: Judge Personas (Days 1-2)

Each judge will be implemented as a LangGraph node that:

1. Receives the full evidence dict from AgentState.
2. Filters evidence by the dimensions relevant to their expertise.
3. For each dimension, constructs a system prompt encoding their persona:

- **Prosecutor:** Adversarial. Looks for gaps, security flaws, laziness. Uses language like "The engineer failed to..." and "This is a clear violation of..."
- **Defense:** Forgiving. Rewards effort and creative workarounds. Uses language like "Despite the limitation, the engineer demonstrated..." and "Partial credit is warranted because..."
- **TechLead:** Pragmatic. Focused on architectural soundness, maintainability, and practical viability. Uses language like "The architecture is modular enough to..."

4. Invokes the LLM with `.with_structured_output(JudicialOpinion)` so the output is guaranteed to be valid JSON matching the Pydantic schema.
5. Includes retry logic (up to 3 attempts) if the LLM returns malformed output.

Phase 2: Chief Justice Synthesis (Days 2-3)

The `ChiefJustice` node will use **deterministic Python logic** — not an LLM prompt:

```
# Rule of Security: confirmed security flaws cap score

if prosecutor_found_security_flaw and evidence_confirms_flaw:

    final_score = min(final_score, 3)

# Rule of Evidence: facts overrule opinions

if defense_claims_metacognition and not evidence_supports_claim:

    defense_opinion_overruled = True

# Variance Re-evaluation: score spread > 2

if max_score - min_score > 2:

    trigger_re_evaluation(dimension_id)
```

This implements **Dialectical Synthesis** — not via an LLM averaging scores, but through explicit rules that weigh evidence over opinion, penalise confirmed security violations, and require dissent summaries when judges disagree significantly.

Phase 3: Report Rendering (Day 3)

The final output will be a structured Markdown report containing:

- **Executive Summary** — overall assessment in 2-3 paragraphs
- **Criterion Breakdown** — per-dimension scores with Prosecutor, Defense, and TechLead arguments, plus the Chief Justice's ruling
- **Dissent Summary** — for any criterion where score variance > 2
- **Remediation Plan** — specific, actionable improvements

Phase 4: VisionInspector & Conditional Edges (Day 4)

- VisionInspector will extract images via PyMuPDF and classify them using a vision-capable LLM (GPT-4o).
- Conditional edges will handle missing evidence gracefully (e.g., if no PDF is provided, skip DocAnalyst and VisionInspector).

3.3 Anticipated Failure Modes

The following failure modes have been identified for the planned judicial layer work. Each is concrete enough that another engineer could reproduce the failure in a test:

Failure Mode	**Trigger**	**Mitigation**
Persona convergence	GPT-4o ignores the adversarial/forgiving framing and produces similar scores and arguments across all three judges	Temperature tuning (Prosecutor: 0.9, Defense: 0.7, TechLead: 0.3); if variance < 0.5 across all dimensions, trigger a re-prompt with an explicit instruction: "Your score must differ from the previous judge's by at least 1 point"
Structured output failure	Judge LLM returns `{"score": "four", "argument": "..."}` — score as string instead of int	`with_structured_output(JudicialOpinion, strict=True)` raises `ValidationError`; 3-attempt retry loop; on third failure, fallback to `score=0` + log for manual review
Cited evidence hallucination	Judge cites a dimension ID that does not exist in	`JudicialOpinion.cited_evidence` is validated against

Failure Mode	Trigger	Mitigation
	evidences (e.g., "cit_evidence": ["nonexistent_dim_id"])	AgentState.evidences.keys() post-construction; invalid citations are stripped with a warning
ChiefJustice rules conflict	Security override (cap score at 3) conflicts with Defense's forgiving ruling on the same dimension	Rules are applied in precedence order: Security Override > Fact Supremacy > Functionality Weight > Defense arguments; this order is hardcoded, not resolved by LLM
Re-evaluation infinite loop	Re-evaluation after high variance produces the same scores again, triggering another re-evaluation	Re-evaluation is triggered at most once per dimension; if variance remains > 2 after re-evaluation, ChiefJustice records the dissent and proceeds with the median score

4. Metacognition — The System Evaluating Itself

The Automaton Auditor is designed to be **self-aware** in a architectural sense. The rubric JSON (rubric/week2_rubric.json) serves as the system's "constitution" — loaded at runtime and distributed to agents based on their target_artifact capability. This means:

- The system can audit **itself** by pointing at its own repository URL.
- Updating the rubric JSON updates the evaluation criteria without code changes.
- The Chief Justice's deterministic rules are derived from the synthesis_rules block in the rubric, making the adjudication logic transparent and auditable.

This is **Metacognition** in practice: the system contains a formal specification of what "good" looks like, uses agents to evaluate against that specification, and resolves disagreements through explicit rules rather than opaque LLM averaging.

5. File Mapping

File	Purpose
src/state.py	Pydantic Evidence, JudicialOpinion, CriterionVerdict, AuditReport; AgentState TypedDict with Annotated reducers
src/tools/repo_tools.py	Sandboxed clone_repo_sandboxed(), extract_git_log(), AST analysis (find_class_definitions, find_stategraph_builder, scan_for_security_issues)
src/tools/doc_tools.py	ingest_pdf(), keyword_search(), extract_mentioned_paths(), search_context() (RAG-lite)
src/nodes/detectives.py	repo_investigator(), doc_analyst(), evidence_aggregator() — all as LangGraph nodes
src/graph.py	build_detective_graph() with parallel fan-out/fan-in and MemorySaver checkpointing
src/nodes/judges.py	Stubbed: prosecutor_node(), defense_node(), tech_lead_node()
src/nodes/justice.py	Stubbed: chief_justice_node() with documented deterministic rules
rubric/week2_rubric.json	Full machine-readable rubric (10 dimensions, synthesis rules)
main.py	CLI entry point: python main.py <repo_url> [--pdf <path>]