AUTOMATON AUDITOR

Architecture & Progress Report

TRP1 Challenge Week 2 — Digital Courtroom System(The Automation Auditor)

Interim Submission

BY : Sumeya Sirmula

Table of Contents

1. Executive Summary

This report documents the architectural decisions, implementation progress, and planned work for the Automaton Auditor system. The system is a hierarchical multi-agent application built on LangGraph that audits peer code submissions using a Digital Courtroom pattern: forensic evidence collection, dialectical judicial evaluation, and deterministic synthesis.

The report covers three areas:

- Architecture decisions made so far, with rationale for each choice (Pydantic over dicts, AST over regex, sandboxing strategy, state management)
- Known gaps in the judicial layer and synthesis engine, with a concrete implementation plan
- Diagrams showing the planned StateGraph flow including detective fan-out/fan-in and judge deliberation patterns

2. Architecture Decisions

2.1 Why Pydantic Over Plain Dicts

Decision

All inter-agent data structures (Evidence, JudicialOpinion) are defined as Pydantic BaseModel classes, not plain Python dictionaries.

Rationale

- Type safety at the boundary: Pydantic validates field types, ranges (confidence: 0.0–1.0), and required fields at construction time. A plain dict with a missing key silently propagates None through the pipeline until it causes a cryptic error three nodes later.
- LLM output enforcement: LangChain's .with_structured_output(PydanticModel) forces the LLM to return a valid instance. Without Pydantic, the LLM returns free-form JSON that may have wrong types, extra fields, or missing keys. Pydantic catches these before they enter state.
- Self-documenting contracts: Each model's Field descriptions serve as documentation and as instructions for the LLM. The description parameter is passed to the model during structured output generation, improving output quality.
- Serialization: Pydantic models serialize cleanly to JSON for LangSmith traces, audit reports, and debugging. Plain dicts require manual serialization logic.

Trade-off Acknowledged

Pydantic adds a dependency and slight overhead. For this system the overhead is negligible compared to LLM API latency (seconds vs microseconds). The safety benefits outweigh the cost.

Aspect	Plain Dict	Pydantic BaseModel
Type validation	None at runtime	Automatic on construction
Missing field	Silent KeyError later	Immediate ValidationError
LLM integration	Manual JSON parsing	.with_structured_output() native
Documentation	Separate docstrings	Field descriptions in schema
Serialization	Manual	.model_dump_json() built-in

2.2 Why AST Parsing Over Regex

Decision

All code structure verification (StateGraph usage, Pydantic models, parallel edges, security checks) uses Python's ast module, not regular expressions.

Rationale

- Structural understanding: AST distinguishes a StateGraph() function call from the string "StateGraph" in a comment, a variable name, or a docstring. Regex cannot make this distinction.
- Argument extraction: AST can extract what arguments were passed to StateGraph(), what classes inherit from BaseModel, and which specific edges were added. Regex can only confirm a pattern exists.
- Rubric compliance: The rubric explicitly scores AST-based analysis as Score 5 (Master Thinker) and regex-based analysis as Score 3 (Competent). This applies both to our agent auditing others AND to how our own code will be audited.
- Composability: AST checks compose naturally. We can walk the tree once and check multiple patterns in a single pass, returning Evidence objects for each finding.

Implementation Structure

Each AST check is a standalone function in src/tools/ast_tools.py that takes a project directory, walks all .py files, and returns one or more Evidence objects:

```
check_stategraph(dir) → Evidence(found=True, location='graph.py:42')
check_pydantic_models(dir) → [Evidence(...), Evidence(...)]
check_parallel_edges(dir) → Evidence(found=True, content='3 fan-out edges')
check_sandboxing(dir) → Evidence(found=True, location='tools/git.py:8')
check_security(dir) → Evidence(found=True, rationale='No os.system calls')
check_structured_output(dir) → Evidence(found=True, location='judges.py:31')
```

Scenario	Regex Result	AST Result
# TODO: add StateGraph	MATCH (false positive)	No match (comment)
name = "StateGraph"	MATCH (false positive)	No match (string literal)
MyStateGraphHelper()	MATCH (partial)	No match (different name)
StateGraph(MyState)	MATCH	MATCH + extracts argument
import StateGraph	MATCH (just import)	Distinguishes import vs call

2.3 Sandboxing Strategy

Decision

All repository operations execute within tempfile.TemporaryDirectory() contexts, using subprocess.run() with shell=False. No os.system(), eval(), or exec() calls.

Rationale

- Automatic cleanup: tempfile.TemporaryDirectory() deletes all contents when the context exits, even on exceptions. A fixed directory accumulates stale clones and requires manual cleanup.
- Isolation: Each audit run gets a fresh, unique directory path. Concurrent runs cannot interfere with each other.
- Shell injection prevention: subprocess.run(["git", "clone", url, dir]) passes arguments as a list, preventing shell injection. os.system("git clone " + url) is vulnerable to URLs containing shell metacharacters.
- Rubric compliance: The rubric checks for TemporaryDirectory usage and penalizes os.system. This is a security-critical scoring criterion.

Implementation Pattern

```
with tempfile.TemporaryDirectory() as tmpdir:
    subprocess.run(['git', 'clone', '--depth', '50', url, tmpdir],
                   capture_output=True, text=True, timeout=60)
    # All analysis happens inside tmpdir
    # Automatically cleaned up on exit
```

2.4 State Management: TypedDict with Annotated Reducers

Decision

The shared agent state uses TypedDict (not a Pydantic model) with Annotated reducers for parallel-write fields.

Rationale

- LangGraph native: StateGraph is designed for TypedDict. While Pydantic can work, TypedDict is the primary documented pattern and avoids serialization edge cases.
- Reducer necessity: Three detectives write to evidences simultaneously. Without operator.ior, only one detective's output survives. Three judges write to opinions simultaneously. Without operator.add, only one judge's opinions survive.
- Minimal state: Fields without parallel writers (repo_url, final_report) use plain types with no reducer, keeping the schema clean.

State Field Map

Field	Type	Reducer	Written By
repo_url	str	None	invoke()
pdf_path	str	None	invoke()
git_commit_hash	str	None	repo_detective
model_metadata	Dict	None	context_builder
rubric_dimensions	List[Dict]	None	context_builder
evidences	Dict[str, List]	operator.ior	3 detectives (parallel)
opinions	List	operator.add	3 judges (parallel)
final_report	str	None	chief_justice

2.5 Per-Layer Model Configuration

Decision

Each layer can use a different LLM model, configured via environment variables and accessed through a factory function get_llm(layer).

Rationale

- Cost optimization: Detectives (Layer 1) primarily use tools, not heavy LLM reasoning. A cheaper model (gpt-4o-mini) suffices. Judges (Layer 2) need strong reasoning for nuanced scoring, justifying a more capable model (gpt-4o).
- Flexibility: Switching from OpenAI to Anthropic or a local model requires changing one env var, not editing multiple source files.
- Instructor guidance: The tutorial explicitly recommended this pattern: “if you can define different LLM per layer, it kind of will help.”

Configuration

```
LAYER1_MODEL=gpt-4o-mini    # Detectives: tool-heavy, cheaper
LAYER2_MODEL=gpt-4o         # Judges: reasoning-heavy, quality
LAYER3_MODEL=gpt-4o         # Chief Justice: optional LLM for report text
```

2.6 Evidence-Opinion Separation

Decision

The Evidence model contains NO score, quality, or recommendation fields. Detectives collect facts. Only judges assign scores.

Rationale

If a detective returns Evidence with a score, that score enters the evidence pool as a "fact." Judges then reason about the score, not the underlying code. This creates circular reasoning: the system judges its own prejudice. By keeping Evidence fact-only (found/not-found, content, location, confidence), all three judges form independent opinions from the same neutral facts. The dialectical process is genuine because the Prosecutor and Defense receive identical evidence and reach different conclusions based on their persona.

2.7 Chief Justice Scope Restriction

Decision

The Chief Justice reads ONLY judicial opinions, NOT raw evidence. It trusts the judges' interpretation.

Rationale

The instructor explicitly stated: "The Supreme Court doesn't really try to look at the evidence. It only looks at the layer two personas." This enforces clean layer separation. The Chief Justice resolves conflicts between judges, not between evidence items. If all judges agree, the Chief Justice simply rubber-stamps.

3. StateGraph Flow Diagrams

3.1 Complete Graph Topology

The following diagram shows the full StateGraph with all nodes, edges, fan-out/fan-in patterns, and superstep groupings. Parallel execution occurs at Supersteps 2 and 4.

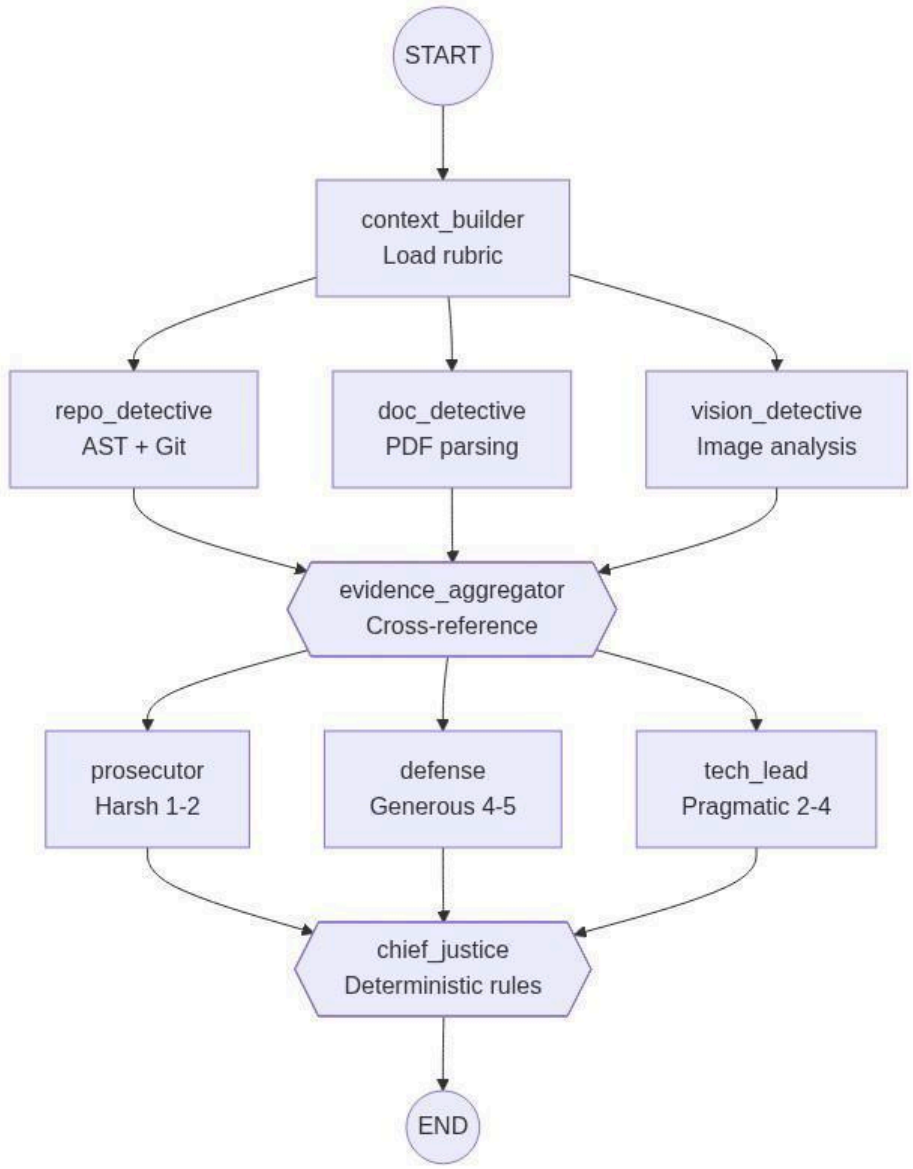

Figure 1: Complete StateGraph topology. Blue nodes = detectives (parallel). Green/Red/Purple nodes = judges (parallel). Orange nodes = sync points. The graph has exactly 5 supersteps.

3.2 Data Flow Across Phases

This diagram shows how state is mutated at each phase, including which reducer handles parallel writes.

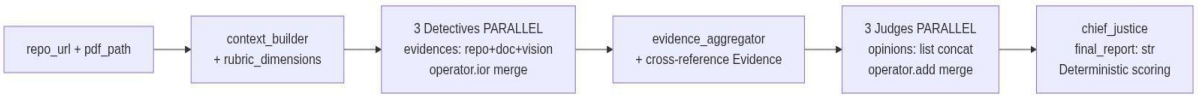

Figure 2: State mutation flow. Phase 1 uses operator.ior to merge evidence dicts. Phase 3 uses operator.add to concatenate opinion lists.

3.3 Edge Wiring Specification

The exact LangGraph edge definitions. Fan-in uses list syntax to ensure the destination runs exactly once.

```
builder.add_edge(START, 'context_builder')

# Fan-out: context_builder → 3 detectives
builder.add_edge('context_builder', 'repo_detective')
builder.add_edge('context_builder', 'doc_detective')
builder.add_edge('context_builder', 'vision_detective')

# Fan-in: 3 detectives → aggregator (JOIN — list syntax)
builder.add_edge(['repo_detective', 'doc_detective', 'vision_detective'], 'evidence_aggregator')

# Fan-out: aggregator → 3 judges
builder.add_edge('evidence_aggregator', 'prosecutor')
builder.add_edge('evidence_aggregator', 'defense')
builder.add_edge('evidence_aggregator', 'tech_lead')

# Fan-in: 3 judges → chief_justice (JOIN — list syntax)
builder.add_edge(['prosecutor', 'defense', 'tech_lead'], 'chief_justice')

builder.add_edge('chief_justice', END)
```

4. Known Gaps & Implementation Plan

4.1 Gap: Judicial Layer (Layer 2)

Current Status

Architecture designed. Prompt templates defined. Factory pattern specified. Not yet implemented.

Specific Gaps

Gap	Risk	Priority
Judge prompt templates not tested against real evidence	Prompts may produce poorly structured arguments or inconsistent scores	HIGH
Structured output parsing not validated	.with_structured_output() may fail on edge cases (empty evidence, ambiguous criteria)	HIGH
Evidence filtering by criterion_id not implemented	Judges may evaluate irrelevant evidence for a given criterion	MEDIUM
Debate cycle (multi-round) not wired	Only affects Master Thinker score. MVP works without it.	LOW

Concrete Plan

- Implement make_judge_node() factory with shared base prompt + 3 persona overlays
- DWire structured output with JudicialOpinion Pydantic model. Test with synthetic evidence.
- Implement evidence filtering: relevant = [e for e in all_evidence if e.criterion_id == criterion['id']]
- Test all 3 judges produce genuinely different scores on the same evidence. Adjust prompts if needed.
- Add a conditional edge for an optional debate round. If time permits, wire round 2 where judges see each other's opinions.

4.2 Gap: Synthesis Engine (Layer 3)

Current Status

Synthesis rules defined as conditional triggers. Report template specified. Not yet implemented.

Specific Gaps

Gap	Risk	Priority
Synthesis rule conditions use string matching on argument text	Fragile if LLM phrases 'security' differently (e.g., 'vulnerability', 'unsafe')	HIGH
Report generation not implemented	Final deliverable depends on this	HIGH
Dissent summary generation	Requires comparing losing vs winning argument per criterion	MEDIUM
Remediation plan extraction	Requires mapping scores < 3 to specific file-level fixes	MEDIUM

Synthesis Rules Decision Tree

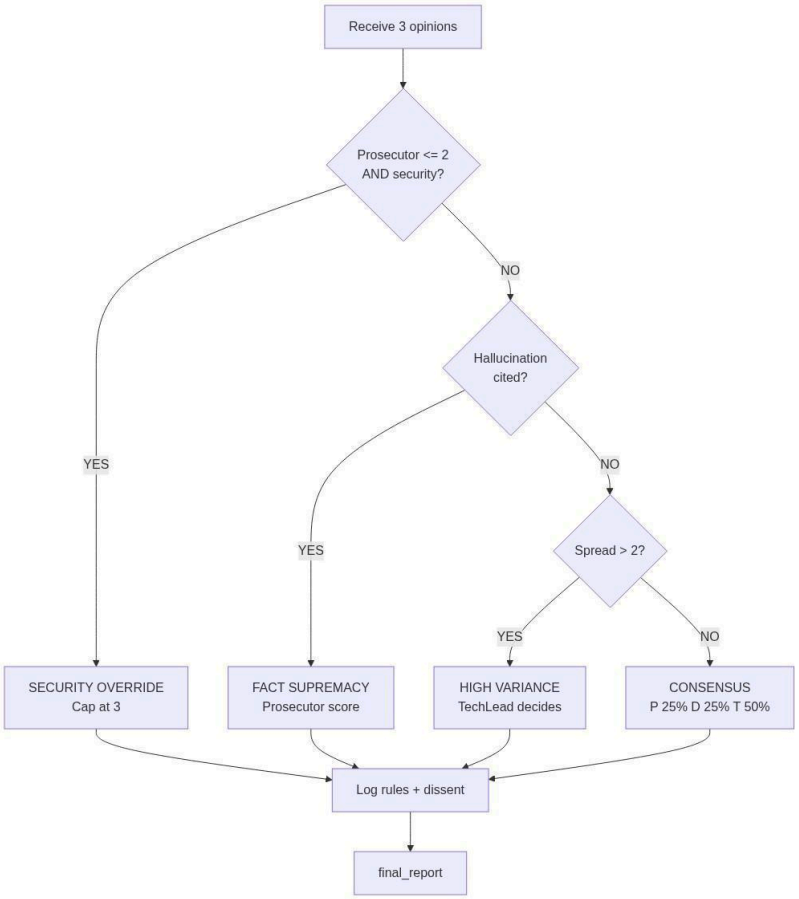

Figure 3: Chief Justice synthesis decision tree. Rules are conditional triggers, not sequential priority. Multiple rules can fire simultaneously.

Concrete Plan

- Implement apply_synthesis_rules() with keyword-based condition matching. Use a keyword set for security detection: {'security', 'vulnerability', 'unsafe', 'injection', 'os.system'}.
- Implement generate_report() that formats all opinions, synthesis results, and dissent into Markdown sections.
- Test with synthetic opinions covering all 4 rule triggers (security, hallucination, high variance, consensus).
- Wire chief_justice node into the graph and run end-to-end tests.

6. Risk Register

Risk	Impact	Likelihood	Mitigation
LLM returns malformed structured output	Judge node fails, superstep rolls back	Medium	Wrap in try/except, retry once, fallback to text parsing
AST parsing fails on non-standard Python	Missing evidence for a criterion	Low	Try/except per file, skip unparseable files with error Evidence
Repo clone times out	No repo evidence collected	Medium	Set 60s timeout, use --depth 50 for shallow clone
PDF has no extractable text (scanned image)	No doc evidence	Low	Detect empty text, return Evidence(found=False)
API rate limits hit during parallel judge calls	One or more judges fail	Medium	Sequential fallback if parallel fails. Log and retry.
Synthesis rules too rigid (keyword matching)	Wrong rule fires or correct rule misses	Medium	Expand keyword sets, add synonym matching, test thoroughly

---

End of Report