AUTOMATON AUDITOR

Architecture Decision Rationale, Gap Analysis, and StateGraph Design

I. Architecture Decision Rationale

The Automaton Auditor is intentionally architected as a **Digital Courtroom system** with layered reasoning:

- **Detective Layer** → Collects factual evidence only
- **Aggregation Layer** → Synchronizes parallel outputs
- **Judicial Layer** → Adversarial scoring via structured opinions
- **Chief Justice** → Deterministic synthesis and resolution

Each major technical decision was chosen to prevent specific architectural failure modes.

1. Typed State: Why Pydantic + TypedDict + Reducers (Not Plain Dicts)

Current Implementation

- `Evidence` and `JudicialOpinion` are Pydantic models.
- `AgentState` is a TypedDict with explicit reducers:
  - `operator.ior` for `evidences`
  - `operator.add` for `opinions`

Why This Was Chosen

Using plain dictionaries would introduce several structural risks in a parallel LangGraph execution environment.

Failure Modes Prevented

1. **Silent schema drift**
   - Plain dicts allow accidental keys or malformed structures.
   - Pydantic enforces strict validation of:
     - `score`
     - `criterion_id`

        - `judge`
        - `cited_evidence`
2. **Parallel state corruption**
    - LangGraph merges branches during fan-in.
    - Without reducers:
        - One branch could overwrite another.
    - With reducers:
        - `opinions` are appended safely.
        - `evidences` are merged safely by bucket.
3. **Malformed LLM output contamination**
    - Judges return JSON.
    - Pydantic validation ensures malformed outputs do not enter system state.

Alternatives Considered

- Plain `dict` discipline → too fragile in multi-agent setting.
- Dataclasses → no built-in validation.
- Marshmallow → unnecessary complexity for internal validation.

Conclusion: Typed state is necessary for deterministic, parallel-safe orchestration.

---

2. AST Parsing vs Regex for Repository Analysis

Current Implementation

- Python `ast` module is used to inspect:
    - `src/graph.py`
    - `src/state.py`

Why AST Was Selected

Regex parsing is text-based and unaware of Python structure.

Failure Modes Prevented

1. **Multiline definition breakage**
   Regex fails on:

```
class AgentState(
    TypedDict,
    total=False
):
```

   AST parses structure correctly.

6. **Nested definitions**
   Regex cannot reliably detect nested classes/functions.
7. **False positives**
   Regex may match commented code or docstrings.
8. **Structure misinterpretation**
   AST reads syntax trees, not raw strings.

Alternatives Rejected

- Regex (too brittle)
- Executing inspected code (unsafe)
- Static analyzers (overkill for scope)

Conclusion: AST guarantees structural correctness over superficial pattern matching.

---

3. Sandboxing Strategy for Cloning Unknown Repositories

Current Implementation

- `tempfile` directories
- No `os.system` usage
- Static inspection only
- No execution of cloned code

Failure Modes Prevented

1. **Arbitrary code execution**
   No cloned code is executed.
2. **Shell injection**
   No direct shell calls.
3. **Filesystem contamination**
   Temporary directory isolation.
4. **Persistent malicious artifacts**
   Temporary directories are disposable.

Alternatives Considered

- Docker isolation → excessive overhead.
- Running repo code → unsafe.

Conclusion: Lightweight sandboxing aligns with static audit design.

---

4. PDF Ingestion: RAG-lite Approach

Current Implementation

- PDF chunking via PyPDF2
- In-memory lexical query
- No embeddings
- No external vector database

Why This Was Chosen

The rubric requires presence/absence verification of conceptual terms.

Failure Modes Prevented

1. **Dependency explosion**
   No vector DB required.
2. **Embedding variance**
   Lexical matching is deterministic.
3. **Cloud cost / API reliance**
   Entirely local.

Trade-Off

- Lower semantic nuance than embeddings.
- Sufficient for rubric keyword validation.

Conclusion: RAG-lite matches scope without overengineering.

---

5. LLM Provider for Judges: Local DeepSeek via Ollama

Current Implementation

- `ChatOllama`
- `deepseek-r1:8b`
- JSON hardening
- `<think>` stripping
- Structured fallback (neutral score)

Why Local LLM

Failure Modes Prevented

1. Cloud rate limits (429)
2. Quota exhaustion
3. Network outages
4. Billing unpredictability

Trade-Off

- Occasional JSON formatting instability
- Slightly slower than high-end cloud APIs

Fallback strategy ensures the system never crashes.

Conclusion: Local LLM prioritizes reproducibility and stability.

II. Gap Analysis and Forward Plan

This section explicitly identifies what is **not yet implemented** and outlines a concrete plan.

Current Gaps

1. Persona Divergence Enforcement

Currently:

- Personas differ by prompt instructions.
- No enforcement ensures divergence in reasoning style.

Planned enhancement, not yet implemented:

- Add divergence scoring heuristics.
- Detect similarity in arguments across judges.
- Force re-evaluation if opinions converge too closely.

---

2. Variance-Based Re-Evaluation in Synthesis

Currently:

- ChiefJustice performs deterministic weighted scoring.

- No variance threshold logic.

Planned enhancement, not yet implemented:

- Compute score variance across judges.
- If variance exceeds threshold:
  - Trigger additional deliberation pass.
  - Or increase TechLead weighting.

---

3. Confidence-Weighted Judging

Currently:

- Judges do not emit confidence values.

Planned enhancement, not yet implemented:

- Extend `JudicialOpinion` with `confidence`.
- Weight final score by confidence.

---

4. JSON Stability Hardening

Occasional fallback occurs due to malformed JSON.

Planned enhancement, not yet implemented:

- Enable Ollama JSON mode if supported.
- Add retry-on-parse-failure.
- Add stricter structural validation pre-Pydantic.

Forward Plan (Sequenced)

Phase 1 — Judicial Robustness

1. Enforce JSON mode.
2. Add retry logic.
3. Add citation validation.

Phase 2 — Dialectical Intelligence

1. Implement variance-based escalation.
2. Introduce dissent analysis summary.
3. Detect persona collapse.

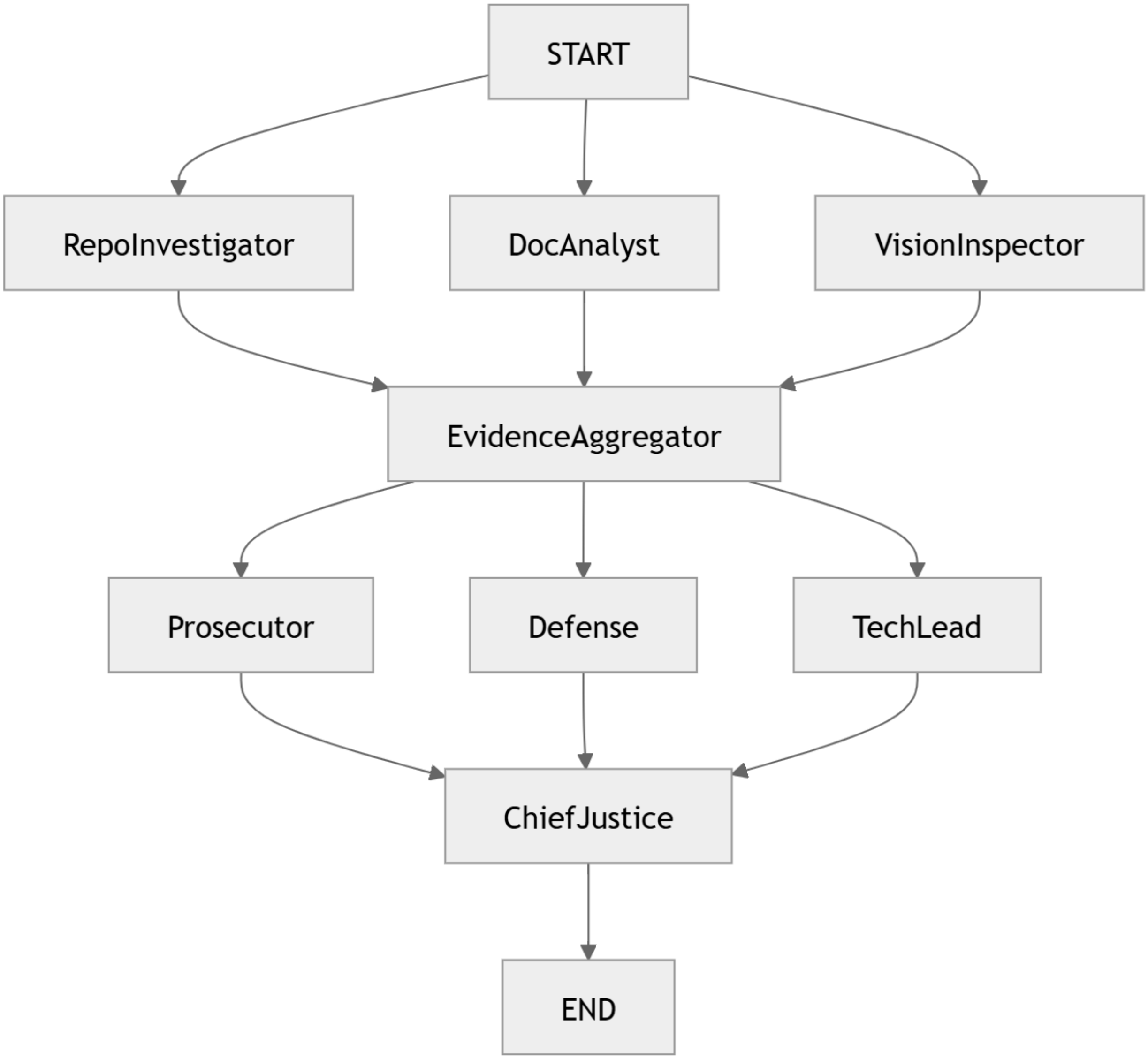
Phase 3 — Observability

1. Log fallback frequency.
2. Track divergence metrics.
3. Track structured output violations.

Each phase is implementable independently.

III. StateGraph Architecture Diagram (Actual Implementation)

This diagram reflects only currently implemented nodes.



Data Flow

- Detectives → `Evidence`
- Aggregator → `Dict[str, List[Evidence]]`
- Judges → `List[JudicialOpinion]`
- ChiefJustice → `final_report: str`

Error Handling (Currently Implemented)

- Repo clone failure → Evidence bucket with failure note.
- Missing PDF → Factual absence.
- Judge JSON failure → Neutral fallback opinions.
- Invalid rubric → Synthetic `rubric_load_failed`.
