Audit Report for https://github.com/nebiyou27/automaton-auditor

Executive Summary: Automated audit completed. See detailed criteria below. **Note:** Judge scores are on a 0 to 10 scale. Final scores are normalised to a 1 to 5 scale. **Overall Score:** 4.20

Criterion: Git Forensic Analysis (git_forensic_analysis)

Final Score: 4 out of 5

Judge Opinions:

- **Defense**: Score 8 out of 10, Argument: Clear progression from setup to tool engineering to graph orchestration.
- **Prosecutor**: Score 8 out of 10, Argument: Clear progression from setup to tool engineering to graph orchestration.
- **TechLead**: Score 10 out of 10, Argument: More than 3 commits showing clear progression from setup to tool engineering to graph orchestration. Atomic, step-by-step history with meaningful commit messages.

Remediation:

To improve Git Forensic Analysis:

- Aim for: More than 3 commits showing clear progression from setup to tool engineering to graph orchestration. Atomic, step-by-step history with meaningful commit messages...
- Avoid: Single 'init' commit or bulk upload of all code at once. No iterative development visible. Timestamps clustered within minutes...
- Next step: Run 'git log --oneline --reverse' on the cloned repository. Count the total number of commits. Check if the commit history tells a progression story: Environment Setup -> Tool Engineering -> Graph Orchestration. Extract all commit messages and timestamps. Flag if there is a single 'init' commit or a 'bulk upload' pattern with no iterative development...

Criterion: State Management Rigor (state_management_rigor)

Final Score: 4 out of 5

Judge Opinions:

- **Defense**: Score 9 out of 10, Argument: TypedDict or BaseModel with Annotated reducers used.
- **Prosecutor**: Score 9 out of 10, Argument: Use of TypedDict or BaseModel with Annotated reducers.

- **TechLead**: Score 10 out of 10, Argument: 'AgentState' uses TypedDict or BaseModel with Annotated reducers. 'Evidence' and 'JudicialOpinion' are Pydantic BaseModel classes with typed fields.

Remediation:

To improve State Management Rigor:

- Aim for: 'AgentState' uses TypedDict or BaseModel with Annotated reducers. 'Evidence' and 'JudicialOpinion' are Pydantic BaseModel classes with typed fields. Reducers like 'operator.add' (for lists) and 'operator.ior' (for dicts) are present...
- Avoid: Plain Python dicts used for state. No Pydantic models. No reducers, meaning parallel agents will overwrite each other's data...
- Next step: Scan for 'src/state.py' or equivalent state definitions in 'src/graph.py'. Use AST parsing (not regex) to find classes inheriting from 'BaseModel' (Pydantic) or 'TypedDict'. Verify that the state actively maintains a collection of 'Evidence' objects and a list of 'JudicialOpinion' objects. Check for the use of 'operator.add' and 'operator.ior' as state reducers in 'Annotated' type hints to prevent data overwriting during parallel execution. Capture the full code snippet of the core 'AgentState' definition...

Criterion: Graph Orchestration Architecture (graph_orchestration)

Final Score: 4 out of 5 Dissent: Significant disagreement among judges.

Judge Opinions:

- **Defense**: Score 7 out of 10, Argument: Conditional edges handle error states.
- **Prosecutor**: Score 10 out of 10, Argument: Two distinct parallel fan-out/fan-in patterns.
- **TechLead**: Score 10 out of 10, Argument: Two distinct parallel fan-out/fan-in patterns: one for Detectives, one for Judges. Conditional edges handle error states. Graph structure: START -> [Detectives in parallel] -> EvidenceAggregator -> [Judges in parallel] -> ChiefJustice -> END.

Remediation:

To improve Graph Orchestration Architecture:

- Aim for: Two distinct parallel fan-out/fan-in patterns: one for Detectives, one for Judges. Conditional edges handle error states. Graph structure: START -> [Detectives in parallel] -> EvidenceAggregator -> [Judges in parallel] -> ChiefJustice -> END...
- Avoid: Purely linear flow (RepoInvestigator -> DocAnalyst -> Judge -> End). No parallel branches. No synchronization node. No conditional edges for error handling...
- Next step: Scan for the 'StateGraph' builder instantiation in 'src/graph.py'. Use AST parsing to analyze 'builder.add_edge()' and 'builder.add_conditional_edges()' calls. Determine if the Detectives (RepoInvestigator, DocAnalyst, VisionInspector) branch out from a single node and run concurrently (fan-out). Verify there is a synchronization node ('EvidenceAggregator' or equivalent) that collects all evidence before the Judges are invoked (fan-in). Verify the Judges (Prosecutor, Defense, TechLead) also fan-out in

parallel from the aggregation node and fan-in before the ChiefJustice. Check for conditional edges that handle 'Evidence Missing' or 'Node Failure' scenarios. Capture the specific Python block defining the graph's nodes and edges...

Criterion: Safe Tool Engineering (safe_tool_engineering)

Final Score: 5 out of 5

Judge Opinions:

- **Defense**: Score 10 out of 10, Argument: All git operations run inside 'tempfile.TemporaryDirectory().'
- **Prosecutor**: Score 9 out of 10, Argument: Use of subprocess.run() with error handling.
- **TechLead**: Score 10 out of 10, Argument: All git operations run inside 'tempfile.TemporaryDirectory()'. 'subprocess.run()' used with error handling. No raw 'os.system()' calls.

Remediation:

To improve Safe Tool Engineering:

- Aim for: All git operations run inside 'tempfile.TemporaryDirectory()'. 'subprocess.run()' used with error handling. No raw 'os.system()' calls. Authentication failures caught and reported...
- Avoid: Raw 'os.system("git clone ")' drops code into the live working directory. No error handling around shell commands. No input sanitization on the repo URL...
- Next step: Scan 'src/tools/' for the repository cloning logic. Verify that 'tempfile.TemporaryDirectory()' or equivalent sandboxing is used for git clone operations. Check for raw 'os.system()' calls – these are a security violation. Verify that 'subprocess.run()' or equivalent is used with proper error handling (capturing stdout/stderr, checking return codes). Ensure the cloned repo path is never the live working directory. Check that git authentication errors are handled gracefully. Capture the specific Python function responsible for executing the repository clone...

Criterion: Structured Output Enforcement (structured_output_enforcement)

Final Score: 4 out of 5

Judge Opinions:

- **Defense**: Score 8 out of 10, Argument: LLMs are invoked using '.with_structured_output()' or '.bind_tools()'.
- **Prosecutor**: Score 8 out of 10, Argument: Use of .with_structured_output() bound to Pydantic JudicialOpinion schema.
- **TechLead**: Score 10 out of 10, Argument: All Judge LLM calls use '.with_structured_output(JudicialOpinion)' or equivalent. Retry logic exists for malformed outputs. Output

is validated against the Pydantic schema before being added to state.

Remediation:

To improve Structured Output Enforcement:

- Aim for: All Judge LLM calls use '.with_structured_output(JudicialOpinion)' or equivalent. Retry logic exists for malformed outputs. Output is validated against the Pydantic schema before being added to state...
- Avoid: Judge nodes call LLMs with plain prompts and parse freeform text responses. No Pydantic validation on output. No retry on parse failure...
- Next step: Scan Judge nodes in 'src/nodes/judges.py'. Verify that LLMs are invoked using '.with_structured_output()' or '.bind_tools()' bound to the Pydantic 'JudicialOpinion' schema. Check that the output includes 'score' (int), 'argument' (str), and 'cited_evidence' (list). Verify there is retry logic or error handling if a Judge returns freeform text instead of structured JSON. Capture the code block responsible for querying the Judge LLMs...

Criterion: Judicial Nuance and Dialectics (judicial_nuance)

Final Score: 5 out of 5

Judge Opinions:

- **Defense**: Score 9 out of 10, Argument: Three clearly distinct personas with conflicting philosophies.
- **Prosecutor**: Score 10 out of 10, Argument: Three clearly distinct personas with conflicting philosophies.
- **TechLead**: Score 10 out of 10, Argument: Three clearly distinct personas with conflicting philosophies. Prompts actively instruct the model to be adversarial (Prosecutor), forgiving (Defense), or pragmatic (Tech Lead). Judges produce genuinely different scores and arguments for the same evidence.

Remediation:

To improve Judicial Nuance and Dialectics:

- Aim for: Three clearly distinct personas with conflicting philosophies. Prompts actively instruct the model to be adversarial (Prosecutor), forgiving (Defense), or pragmatic (Tech Lead). Judges produce genuinely different scores and arguments for the same evidence...
- Avoid: Single agent acts as 'The Grader' with no persona separation. Or three judges exist but share 90% of prompt text, producing near-identical outputs. Scores are random or purely praise/criticism without nuance...
- Next step: Scan 'src/nodes/judges.py' or prompt templates. Verify that Prosecutor, Defense, and Tech Lead personas have distinct, conflicting system prompts. Compare the three prompts – if they share more than 50% of text, flag as 'Persona Collusion'. Check if the Prosecutor prompt includes adversarial language and instructions to look for gaps, security flaws, and laziness. Check if the Defense prompt includes instructions to reward effort, intent, and creative workarounds. Check if the Tech Lead prompt focuses on

architectural soundness, maintainability, and practical viability. Verify the graph forces all three judges to run in parallel on the same evidence for each criterion...

Criterion: Chief Justice Synthesis Engine (chief_justice_synthesis)

Final Score: 4 out of 5

Judge Opinions:

- **Defense**: Score 8 out of 10, Argument: Deterministic Python if/else logic implementing named rules.
- **Prosecutor**: Score 9 out of 10, Argument: Deterministic Python if/else logic implementing named rules.
- **TechLead**: Score 10 out of 10, Argument: Deterministic Python if/else logic implementing named rules (security override, fact supremacy, functionality weight). Score variance triggers specific re-evaluation. Output is a Markdown file with Executive Summary, Criterion Breakdown (with dissent), and Remediation Plan.

Remediation:

To improve Chief Justice Synthesis Engine:

- Aim for: Deterministic Python if/else logic implementing named rules (security override, fact supremacy, functionality weight). Score variance triggers specific re-evaluation. Output is a Markdown file with Executive Summary, Criterion Breakdown (with dissent), and Remediation Plan...
- Avoid: ChiefJustice is just another LLM prompt that averages the three judge scores. No hardcoded rules. No dissent summary. Output is console text or unstructured...
- Next step: Scan 'src/nodes/justice.py' for the ChiefJusticeNode implementation. Verify the conflict resolution uses hardcoded deterministic Python logic, not just an LLM prompt. Check for these specific rules: (1) Rule of Security – if the Prosecutor identifies a confirmed security vulnerability, the score is capped at 3 regardless of Defense arguments. (2) Rule of Evidence – if the Defense claims 'Deep Metacognition' but Detective evidence shows the artifact is missing, the Defense is overruled. (3) Rule of Functionality – if the Tech Lead confirms the architecture is modular, this carries the highest weight for the Architecture criterion. Check if score variance > 2 triggers a specific re-evaluation rule. Verify the output is a structured Markdown report, not a console print...

Criterion: Theoretical Depth (Documentation) (theoretical_depth)

Final Score: 4 out of 5 Dissent: Significant disagreement among judges.

Judge Opinions:

- **Defense**: Score 7 out of 10, Argument: Terms appear in detailed architectural explanations.
- **Prosecutor**: Score 8 out of 10, Argument: Terms appear in detailed architectural explanations.

- **TechLead**: Score 10 out of 10, Argument: Terms appear in detailed architectural explanations. The report explains how Dialectical Synthesis is implemented via three parallel judge personas.

Remediation:

To improve Theoretical Depth (Documentation):

- Aim for: Terms appear in detailed architectural explanations. The report explains how Dialectical Synthesis is implemented via three parallel judge personas. Fan-In/Fan-Out is tied to specific graph edges. Metacognition is connected to the system evaluating its own evaluation quality…
- Avoid: Terms appear only in the executive summary or introduction. No connection to actual implementation. 'We used Dialectical Synthesis' with no explanation of how…
- Next step: Search the PDF report for these specific terms: 'Dialectical Synthesis', 'Fan-In / Fan-Out', 'Metacognition', 'State Synchronization'. Determine if the term appears in a substantive architectural explanation or is just a buzzword dropped in the executive summary. Check if the report explains HOW the architecture executes these concepts, not just that they exist. Flag terms that appear without supporting explanation as 'Keyword Dropping'…

Criterion: Report Accuracy (Cross-Reference) (report_accuracy)

Final Score: 4 out of 5

Judge Opinions:

- **Defense**: Score 9 out of 10, Argument: All file paths mentioned in the report exist in the repo.
- **Prosecutor**: Score 9 out of 10, Argument: All file paths mentioned in the report exist in the repo.
- **TechLead**: Score 10 out of 10, Argument: All file paths mentioned in the report exist in the repo. Feature claims match code evidence. Zero hallucinated paths.

Remediation:

To improve Report Accuracy (Cross-Reference):

- Aim for: All file paths mentioned in the report exist in the repo. Feature claims match code evidence. Zero hallucinated paths…
- Avoid: Report references files that do not exist. Claims parallel execution but code shows linear flow. Multiple hallucinated paths detected…
- Next step: Extract all file paths mentioned in the PDF report (e.g., 'We isolated the AST logic in src/tools/ast_parser.py', 'We implemented parallel Judges in src/nodes/judges.py'). Cross-reference each claimed file path against the evidence collected by the RepoInvestigator. Build two lists: (1) Verified Paths – files that the report mentions and actually exist in the repo. (2) Hallucinated Paths – files the report claims exist but the RepoInvestigator found no evidence of. Flag any claims about features (e.g., 'We implemented parallel Judges') where the code evidence contradicts the claim…

Criterion: Architectural Diagram Analysis (swarm_visual)

Final Score: 4 out of 5

Judge Opinions:

- **Defense**: Score 8 out of 10, Argument: Diagram accurately represents the StateGraph with clear parallel branches.
- **Prosecutor**: Score 10 out of 10, Argument: Diagram accurately represents the StateGraph with clear parallel branches.

Remediation:

To improve Architectural Diagram Analysis:

- Aim for: Diagram accurately represents the StateGraph with clear parallel branches for both Detectives and Judges. Fan-out and fan-in points are visually distinct. Flow matches the actual code architecture…
- Avoid: Generic box-and-arrow diagram with no indication of parallelism. Or no diagram present at all. Diagram shows linear flow that contradicts the parallel architecture claimed in the report…
- Next step: Extract images from the PDF report. Classify each diagram: is it an accurate LangGraph State Machine diagram, a sequence diagram, or just generic flowchart boxes? Check if the diagram explicitly visualizes the parallel split: START -> [Detectives in parallel] -> Evidence Aggregation -> [Prosecutor || Defense || TechLead in parallel] -> Chief Justice Synthesis -> END. Verify the diagram distinguishes between parallel branches and sequential steps. Flag diagrams that show a simple linear pipeline as 'Misleading Architecture Visual'…

Remediation Plan

Apply remediation steps per criterion to improve architecture and compliance.

VisionInspector Status

No visual evidence considered (disabled or none found).