FDE Challenge Week 2: The Automaton Auditor

Orchestrating Deep LangGraph Swarms for Autonomous Governance

The Business Objective

The Paradigm Shift: From Generation to Governance In Week 1, you learned to manage a single "Silicon Worker." Now, we face the scaling problem.

In a mature AI-Native Enterprise, the volume of code generated by autonomous agents will outpace human review capacity by orders of magnitude. If 1,000 agents are generating features concurrently, humans cannot manually review every pull request. The bottleneck shifts from generating code to evaluating it.

The Solution: Automated Quality Assurance Swarms To maintain quality at scale, we must engineer Automated Auditor Swarms: deep, multi-agent systems capable of:

1. **Forensic Analysis:** objectively verifying the existence and structure of code artifacts.
2. **Nuanced Judgment:** applying complex rubrics that require interpretation (e.g., "Is this architecture modular?").
3. **Constructive Feedback:** providing actionable remediation steps, not just pass/fail grades.

The Meta-Game (MinMax Optimization): This week, you will build the very system that judges you.

1. **Build:** You will architect a Deep LangGraph Agent designed to grade Week 2 submissions (the auditor itself) based on a strict "Forensic Expert" rubric.
2. **Deploy:** You will unleash your agent to audit the Week 2 repository of your peers to test if their code works and identify gaps in their implementation.
3. **Refine:** Simultaneously, your peers' agents will audit your work. You will use their automated feedback to fix bugs in your Week 2 repository (improving your score) and refine your Week 2 agent to catch issues you missed (improving your grading capability). This creates an adversarial "MinMax" loop where the auditor improves the auditor.

Your Mission:

You are building a **Digital Courtroom**.

- **Input:** A single GitHub Repository URL and a PDF Report.

- **Process:** A hierarchical swarm of "Detective" agents collects evidence, passes it to "Judge" agents with distinct personas (Prosecutor, Defense, Tech Lead), and a "Chief Justice" synthesizes a final verdict.
- **Output:** A production-grade Audit Report that stands up to scrutiny.

Applicability: This architecture is not just for grading homework. It is the blueprint for:

- **Automated Security Audits:** Swarms that hunt for vulnerabilities in every PR.
- **Compliance Governance:** Agents that ensure ISO/SOC2 compliance in real-time.
- **Architectural Review:** Systems that prevent "Spaghetti Code" before it merges.

Mandatory Research & Conceptual Foundation

Core Concepts:

- **Deep Agents & Orchestration:**
    - LangGraph Documentation (Master the StateGraph, nodes, parallel execution, and conditional_edges).
    - Multi-Agent Systems (MAS) - Understanding how specialized agents collaborate to solve problems single LLMs cannot.
- **The Forensic Mindset:**
    - **Objective Evidence vs. Subjective Opinion:** Your agents must distinguish between facts (file exists) and judgments (code is clean).
    - **Review the Week 2 Rubric (attached below).** This is your agent's "Constitution." Your agents must be programmed to execute the specific "Evidence Collection Instructions" defined there.
- **Critique & Evaluation:**
    - LLMs as Judges - How to prompt models to evaluate other models effectively.
    - Constitutional AI - Using a set of principles (the rubric) to guide AI behavior.

Why This Matters: Building an evaluator is harder than building a generator. It requires Metacognition: the ability to think about thinking. You are building a system that understands quality.

The Architecture: The Digital Courtroom

You must implement a **Hierarchical State Graph** using LangGraph. A single LLM cannot do this job; it requires specialized roles.

Layer 1: The Detective Layer (Forensic Sub-Agents)

These agents **do not opinionate**. They only collect facts based on strict forensic protocols. Their output is a structured JSON Evidence object, untainted by bias.

- **RepoInvestigator (The Code Detective):**

        - **Tools:** git clone, git log, file_read, ast_parse (using Python's ast module or tree-sitter).
        - **Forensic Protocol A (State Structure):** Verify the existence of typed state. Does src/state.py or src/graph.py exist? Do they contain valid Pydantic BaseModel or TypedDict schemas?
        - **Forensic Protocol B (Graph Wiring):** Do not just check for the string "StateGraph". You must verify if the graph is functionally wired for parallelism. Does the AST reveal builder.add_edge() creating a fan-out architecture?
        - **Forensic Protocol C (The Git Narrative):** Specific analysis of git log. Is the history atomic (step-by-step) or monolithic (one "init" commit)? Extract the timestamps.
- **DocAnalyst (The Paperwork Detective):**
    - **Tools:** pdf_parse, markdown_read, cross_reference.
    - **Forensic Protocol A (Citation Check):** When the report claims "We implemented parallel Judges in src/nodes/judges.py", this agent must cross-reference RepoInvestigator's data. Does that file actually exist? If the report cites a non-existent file, flag as "Hallucination."
    - **Forensic Protocol B (Concept Verification):** Scan for deep understanding of "Dialectical Synthesis" or "Metacognition." Does the text just drop the keyword, or does it explain how the architecture executes it?
- **VisionInspector (The Diagram Detective):**
    - **Tools:** image_analysis (Gemini Pro Vision/GPT-4o).
    - **Forensic Protocol A (Flow Analysis):** Analyze architectural diagrams. Does the arrow flow clearly from Detectives (Parallel) -> Evidence Aggregation -> Judges (Parallel) -> Synthesis? Or is it a simple linear pipeline?

Layer 2: The Judicial Layer (The Dialectical Bench)

The Judicial Layer applies point assignments of a rubric by doing criterion-by-criterion analysis through distinct persona lenses, ensuring the "Digital Courtroom" operates on a dialectical model (Thesis-Antithesis-Synthesis).

This is the core of the reasoning engine. You must not simply feed evidence to a generic "Grader." Instead, you must implement a **Dialectical Process** where three distinct personas analyze the **same evidence** for **each rubric criterion** independently.

For every dimension of the Week 2 Rubric (e.g., "LangGraph Architecture", "Judicial Nuance"), all three judges must submit an opinion based on their specific lens.

- **The Prosecutor (The Critical Lens)**
    - **Core Philosophy:** "Trust No One. Assume Vibe Coding."
    - **Objective:** Scrutinize the evidence for gaps, security flaws, and laziness.
    - **Prompting Strategy:** If the rubric asks for "Parallel Orchestration" and the evidence shows "Linear pipeline," you must argue for a Score of 1. Look specifically for bypassed structure. If Judges return freeform text instead

of Pydantic models, charge the defendant with "Hallucination Liability." Provide a harsh score and a list of specific missing elements.

- **The Defense Attorney (The Optimistic Lens)**
    - **Core Philosophy:** "Reward Effort and Intent. Look for the 'Spirit of the Law'."
    - **Objective:** Highlight creative workarounds, deep thought, and effort, even if the implementation is imperfect.
    - **Prompting Strategy:** If the code is buggy but the architecture report shows deep understanding of LangGraph state reducers, argue that the student matches the "Master Thinker" profile despite syntax errors. Look at the Git History evidence. If the commits tell a story of struggle and iteration, argue for a higher score based on "Engineering Process." Provide a generous score and highlight strengths.
- **The Tech Lead (The Pragmatic Lens)**
    - **Core Philosophy:** "Does it actually work? Is it maintainable?"
    - **Objective:** Evaluate architectural soundness, code cleanliness, and practical viability.
    - **Prompting Strategy:** Ignore the "Vibe" and the "Struggle." Focus on the Artifacts. Is the operator.add reducer actually used to prevent data overwriting? Are the tool calls isolated and safe? You are the tie-breaker. If the Prosecutor says "1" (Security flaw) and Defense says "5" (Great effort), you assess the Technical Debt. Provide a realistic score (1, 3, or 5) and technical remediation advice.

The Judicial Workflow: For *each* Rubric Criterion (e.g., "LangGraph Architecture"):

1. **State Input:** Evidence object (e.g., "Graph builds linearly, no parallel branches detected").
2. **Parallel Execution:**
    - Prosecutor: "Graph orchestration is fundamentally flawed and misses the core requirement. Score: 1."
    - Defense: "The state management is sound even if the routing is simple. Score: 3."
    - Tech Lead: "Linear execution creates a bottleneck. Fails architectural standard. Score: 1."
3. **Output:** A list of JudicialOpinion objects containing three conflicting views.

Layer 3: The Supreme Court (Final Verdict)

This node is the **Synthesis Engine**. It does not merely average the scores; it resolves the dialectical conflict generated by Layer 2 to produce a final, actionable ruling.

- **ChiefJusticeNode:**
    - **Input:** The JudicialOpinion objects (Prosecutor, Defense, Tech Lead arguments) for every criterion.
    - **Deliberation Protocol (Hardcoded Rules):**
        - *Rule of Security:* If The Prosecutor identifies a confirmed security vulnerability (e.g., os.system with unsanitized inputs), this overrides

any "Effort" points from the Defense. Security flaws cap the score at 3.
    - *Rule of Evidence:* If The Defense claims "Deep Metacognition" but RepoInvestigator found no PDF report, the Defense is overruled for hallucination.
    - *Rule of Functionality:* If The Tech Lead confirms the architecture is modular and workable, this carries the highest weight for the "Architecture" criterion.
- **Output Generation (The Audit Report):** The node must generate a structured Markdown report containing:
    - **The Verdict:** Final Score (1-5) per criterion.
    - **The Dissent:** A summary of the conflict (e.g., "The Defense argued for code effort, but the Prosecutor correctly noted that the graph fails to compile due to missing state reducers.").
    - **The Remediation Plan:** Specific, file-level instructions for the trainee.

Implementation Curriculum

Note: This is not a "Toy Model" exercise. You are building Production-Grade Infrastructure.

Phase 1: The Production Environment (Infrastructure)

Objective: Establish a typed, observable, and isolated runtime environment. A simple script is insufficient; you need a robust StateGraph.

1. **State Definition (Pydantic):** Do not use simple Python dicts. Define your AgentState using Pydantic models and TypedDict to enforce strict typing and proper state reduction.

```python
import operator
from typing import Annotated, Dict, List, Literal, Optional

from pydantic import BaseModel, Field
from typing_extensions import TypedDict


# --- Detective Output ---


class Evidence(BaseModel):
    goal: str = Field()
    found: bool = Field(description="Whether the artifact exists")
    content: Optional[str] = Field(default=None)
    location: str = Field(
        description="File path or commit hash",
    )
    rationale: str = Field(
        description="Your rationale for your confidence "
        "on the evidence you find for this particular goal",
    )
    confidence: float


# --- Judge Output ---


class JudicialOpinion(BaseModel):
    judge: Literal["Prosecutor", "Defense", "TechLead"]
    criterion_id: str
    score: int = Field(ge=1, le=5)
    argument: str
    cited_evidence: List[str]


# --- Chief Justice Output ---


class CriterionResult(BaseModel):
    dimension_id: str
    dimension_name: str
    final_score: int = Field(ge=1, le=5)
    judge_opinions: List[JudicialOpinion]
    dissent_summary: Optional[str] = Field(
        default=None,
        description="Required when score variance > 2",
    )
    remediation: str = Field(
        description="Specific file-level instructions "
        "for improvement",
    )


class AuditReport(BaseModel):
    repo_url: str
    executive_summary: str
    overall_score: float
    criteria: List[CriterionResult]
    remediation_plan: str


# --- Graph State ---


class AgentState(TypedDict):
    repo_url: str
    pdf_path: str
    rubric_dimensions: List[Dict]
    # Use reducers to prevent parallel agents
    # from overwriting data
```

```
evidences: Annotated[
    Dict[str, List[Evidence]], operator.ior
]
opinions: Annotated[
    List[JudicialOpinion], operator.add
]
final_report: AuditReport
```

2. **Environment Isolation:** Use the uv python package manager. Strictly manage dependencies. You must handle API keys securely (never hardcoded) using .env.
3. **Observability:** Integrate LangSmith tracing immediately. Set LANGCHAIN_TRACING_V2=true. You will need to debug complex multi-agent chains; console logs will not suffice.

Phase 2: Advanced Tool Engineering (The Detective Layer)

Objective: Build forensic tools that don't just "read text" but "understand structure."

1. **RepoInvestigator (The AST Detective):**
    - **Logic:** Do not rely on Regex. It is brittle. Use Python's built-in ast module or a robust parser to verify if classes like StateGraph are syntactically valid and instantiated. You may use the gitingest online or python package to parse and distill github repositories.
    - **Safety:** Your agent is cloning unknown code. Run git commands in a sandboxed temporary directory (tempfile). Handle git authentication errors gracefully.
    - **Task:** Implement analyze_graph_structure(path: str) and extract_git_history(path: str).
2. **DocAnalyst (The Context Detective):**
    - **Logic:** Implement a "RAG-lite" approach. The PDF report might be large. Don't dump the whole text into context. You may use the Docling Python package to parse PDFs.
    - **Task:** Implement ingest_pdf(path: str) that chunks the document and allows the agent to query: "What does the report say about Dialectical Synthesis?"
3. **VisionInspector (The Multimodal Detective):**
    - **Logic:** Use Multimodal LLMs. You must handle image extraction from the PDF.
    - **Task:** Implement extract_images_from_pdf(path: str) and pass them to the vision model with specific questions: "Is this a StateGraph diagram or a generic box diagram?"
    - **Scope:** You may implement this feature, but running it to get results is optional.

Phase 3: Orchestrating the Bench (The Judicial Layer)

Objective: Force the LLM to adhere to the "Digital Courtroom" protocol without hallucination.

1. **Structured Output Enforcement:**
    - Use .bind_tools() or .with_structured_output() for the Judges. They **must** return a JSON object containing score (int), reasoning (str), and citations (list).
    - **Rule:** If a Judge returns free text, the node acts as a parser error and forces a retry.
2. **Graph Construction:**
    - **Parallelism:** The Detectives (Repo, Doc, Vision) must run in parallel branches.
    - **Fan-In:** Implement a synchronization node (EvidenceAggregator) that collects all JSON evidence before waking the Judges.
    - **Fan-Out:** The Judges (Prosecutor, Defense, TechLead) must run in parallel, analyzing the *same* evidence independently.
3. **The Constitution:**
    - Your System Prompts must dynamically load the **Week 2 Rubric**. You must provide a rubric.json file so your agent can dynamically integrate the scoring rules into its reasoning loop.

Phase 4: The Supreme Court & Feedback Loop

Objective: Synthesize conflict and operationalize the swarm.

1. **The Synthesis Engine:**
    - Implement the ChiefJusticeNode. It needs a "Conflict Resolution Strategy."
    - **Logic:** Hardcode deterministic Python logic to resolve disputes. If variance in scores > 2 (e.g., Prosecutor says 1, Defense says 5), trigger a rule set to re-evaluate specific evidence based on the JSON configuration.
2. **Report Generation:**
    - The final output must be a Markdown file, not a console print.
    - It must follow the structure: **Executive Summary -> Criterion Breakdown -> Remediation Plan**.

Deliverables

Interim Submission - Wednesday 21hr UTC

PDF Report

- Architecture decisions made so far (why Pydantic over dicts, how AST parsing was structured, sandboxing strategy)

- Known gaps and a concrete plan for the judicial layer and synthesis engine
- Diagrams showing the planned StateGraph flow (detective fan-out/fan-in)

Submit a GitHub Repository containing:

- src/state.py -- Pydantic/TypedDict state definitions (Evidence, JudicialOpinion, AgentState) with proper reducers (operator.add, operator.ior)
- src/tools/repo_tools.py -- Sandboxed git clone (using tempfile), git log extraction, AST-based graph structure analysis
- src/tools/doc_tools.py -- PDF ingestion and chunked querying (RAG-lite approach)
- src/nodes/detectives.py -- RepoInvestigator and DocAnalyst implemented as LangGraph nodes that output structured Evidence objects
- src/graph.py -- Partial StateGraph wiring the detectives in parallel (fan-out) with an EvidenceAggregator node (fan-in). Judges not required yet.
- pyproject.toml -- Locked dependencies managed via uv
- .env.example -- Lists all required API keys and environment variables (no actual secrets committed)
- README.md -- Setup instructions, how to install dependencies, and how to run the detective graph against a target repo URL
- reports/interim_report.pdf -- Your PDF report committed to the repo so peers can access it

Peer-Gradable GitHub Repository:

- This should be the same repository you submitted above. It will be used by your assigned peer to review and grade your work using their agent.

Final Submission - Sunday 03hr UTC

PDF Report

- Executive summary
- Architecture deep dive explaining Dialectical Synthesis, Fan-In/Fan-Out, and Metacognition with substance (not buzzwords)
- Architectural diagrams (StateGraph visualization showing parallel flow)
- Criterion-by-criterion breakdown of self-audit results
- Reflection on the MinMax feedback loop:
    - What your peer's agent caught that you missed
    - How you updated your agent to detect similar issues in others
- Remediation plan for remaining gaps

Submit a GitHub Repository containing:

Source Code (everything from interim, refined and extended):

- src/state.py -- Finalized state definitions.
- src/tools/repo_tools.py -- Finalized forensic tools for repo analysis
- src/tools/doc_tools.py -- Finalized PDF parsing and cross-referencing tools
- src/nodes/detectives.py -- RepoInvestigator, DocAnalyst, and VisionInspector (VisionInspector: implementation required, execution optional)
- src/nodes/judges.py -- Prosecutor, Defense, and Tech Lead with distinct system prompts. Must use .with_structured_output() or .bind_tools() bound to the JudicialOpinion Pydantic schema
- src/nodes/justice.py -- ChiefJusticeNode with hardcoded deterministic conflict resolution rules (security override, fact supremacy, dissent requirement). Produces an AuditReport which gets serialized to Markdown as the final_report in AgentState.
- src/graph.py -- Complete StateGraph with parallel fan-out/fan-in for both detectives and judges, conditional edges for error handling, and end-to-end flow from repo URL input to rendered Markdown report

Infrastructure:

- pyproject.toml -- Locked dependencies managed via uv
- .env.example -- Lists all required API keys and environment variables
- README.md -- Full instructions: setup, dependency installation, and how to run the swarm against any target repo URL and PDF report
- Dockerfile -- (Optional but recommended) Containerized runtime for the auditor

Audit Reports:

- audit/report_onself_generated/ -- Markdown report generated by running your agent against your own Week 2 repo
- audit/report_onpeer_generated/ -- Markdown report generated by running your agent against your assigned peer's Week 2 repo
- audit/report_bypeer_received/ -- The Makdown report your peer's agent generated when auditing your Week 2 repo

Each report is a Markdown serialization of the AuditReport model:
- **Executive Summary** -- overall verdict and aggregate score
- **Criterion Breakdown** -- one section per rubric dimension (10 total), each containing the final score, three judge opinions with cited evidence, and dissent summary where applicable
- **Remediation Plan** -- specific, file-level instructions grouped by criterion

Report:
- reports/final_report.pdf -- Your PDF report committed to the repo so peers' agents can access it during auditing

Langsmith Traces

- Link to your langsmith trace showing the full reasoning loop: detectives collecting evidence, judges arguing, and the Chief Justice synthesizing the verdict

Submit a Video Demonstration (5-min max)

A screen recording demonstrating the full workflow end to end:

- Running the agent against a target repo URL and PDF report
- Detectives collecting evidence (showing structured Evidence output)
- Judges deliberating in parallel (showing distinct opinions from Prosecutor, Defense, and Tech Lead)
- ChiefJustice synthesizing the final verdict
- The rendered Markdown audit report as final output

Automation Auditor Input Rubric

This is the binding law for your agent swarm. Your Detectives must be programmed to execute these specific evidence collection protocols, and your Judges must cite these specific standards when rendering a verdict.

Protocol A: The Forensic Evidence Collection Standards (For Detectives)

1. Instructions for RepoInvestigator (The Code Detective) *Target: The GitHub Repository*

- **Evidence Class: Git Forensic Analysis**
  - **Command:** `git log --oneline --reverse`
  - **Success Pattern:** >3 commits. Progression: Environment Setup -> Tool Engineering -> Graph Orchestration.
  - **Failure Pattern:** Single "init" commit or "bulk upload" of code.
  - **Capture:** List of commit messages and timestamps.
- **Evidence Class: State Management Rigor (Phase 0)**
  - **File Check:** Scan for `src/state.py` or equivalent definitions in `src/graph.py`.
  - **Content Scan (AST):** Look for classes inheriting from `BaseModel` (Pydantic) or `TypedDict`. Does the state actively maintain a collection of `Evidence` and a list of `JudicialOpinion` objects?
  - **Capture:** Code snippet of the core `AgentState` definition.
- **Evidence Class: Graph Orchestration (Phase 1)**
  - **Graph Definition:** Scan for the `StateGraph` builder instantiation.
  - **Parallelism Check:** Use AST parsing to analyze `builder.add_edge()`. Do the Detectives or Judges branch out from a single node and run concurrently (Fan-Out)? Is there a synchronization node (Fan-In) before the Judges are invoked?
  - **Capture:** The specific Python block defining the graph's nodes and edges.
- **Evidence Class: Safe Tool Engineering (Phase 2)**

    - **Git Sandboxing:** Scan src/tools/ for the cloning logic. Do they use tempfile.TemporaryDirectory() for isolation?
    - **Security Enforcement:** Look for raw os.system calls without input sanitization or error handling around standard out/error.
    - **Capture:** The specific Python function executing the repository clone.
- **Evidence Class: Structured Output (Phase 3)**
    - **Enforcement:** Scan Judge nodes (src/nodes/judges.py). Do they invoke LLMs using .with_structured_output() or .bind_tools() bound to the Pydantic JudicialOpinion schema?
    - **Capture:** The code block responsible for querying the Judge LLMs.

2. Instructions for DocAnalyst (The Paperwork Detective) *Target: The PDF Report*

- **Evidence Class: Theoretical Depth**
    - **Keyword Search:** "Dialectical Synthesis", "Fan-In / Fan-Out", "Metacognition", "State Synchronization".
    - **Context Check:** Do these terms appear in architectural explanations, or are they just buzzwords thrown in the executive summary?
    - **Capture:** The specific sentences detailing these orchestration concepts.
- **Evidence Class: Host Analysis Accuracy**
    - **Cross-Reference:** Extract file paths mentioned in the report (e.g., "We isolated the AST logic in src/tools/ast_parser.py").
    - **Verification:** Request confirmation from RepoInvestigator. Do these files actually exist?
    - **Capture:** A definitive list of Hallucinated Paths vs. Verified Paths.

3. Instructions for VisionInspector (The Diagram Detective) *Target: Extracted Images*

- **Evidence Class: The Swarm Visual**
    - **Type Classification:** Is it an accurate LangGraph State Machine diagram, a sequence diagram, or just generic flowchart boxes?
    - **Critical Flow:** Does it explicitly visualize the parallel split: Evidence Aggregation -> (Prosecutor || Defense || TechLead) -> Chief Justice Synthesis?
    - **Capture:** Classification string and structural description of the visualized flow.

Protocol B: The Judicial Sentencing Guidelines (For Judges)

Your Judges must interpret the evidence using these strict precedents. The goal is to weed out single-prompt wrappers masking as agents.

1. The Statute of Orchestration (Prosecutor's Handbook)

- **Violation:** If the StateGraph defines a purely linear flow (e.g., RepoInvestigator -> DocAnalyst -> Judge -> End) instead of parallel fan-out execution.
    - *Charge:* "Orchestration Fraud."
    - *Penalty:* Max Score for "LangGraph Architecture" = 1.
- **Violation:** If Judge nodes return freeform text and lack Pydantic validation for structured JSON output.
    - *Charge:* "Hallucination Liability."
    - *Penalty:* Max Score for "Judicial Nuance" = 2.

2. The Statute of Engineering (Tech Lead's Handbook)

- **Precedent:** "Pydantic Rigor vs. Dict Soups."
    - *Standard:* State definitions and JSON outputs must use typed structures (BaseModel). If standard Python dictionaries are used to pass complex nested state:
    - *Ruling:* "Technical Debt." Score = 3 (Functionally executes but is architecturally brittle and unmaintainable).
- **Precedent:** "Sandboxed Tooling."
    - *Standard:* System-level interactions (cloning, parsing) must be wrapped in error handlers and temporary directories. If os.system('git clone <url>') drops code into the live working directory:
    - *Ruling:* "Security Negligence." Overrides all effort points for the "Forensic Accuracy" criterion.

3. The Statute of Effort (Defense Attorney's Handbook)

- **Mitigation:** If the StateGraph fails to compile due to a minor edge validation error, but the underlying AST parsing logic built for the Detectives is highly sophisticated (e.g., extracting specific function imports without regex).
    - *Argument:* "The engineer achieved deep code comprehension but tripped on framework syntax."
    - *Request:* Boost Score from 1 to 3 for "Forensic Accuracy" despite the broken graph.
- **Mitigation:** If the Chief Justice synthesis node is an LLM prompt instead of hardcoded deterministic rules, but the Judge personas are highly distinct, well-prompted, and actively disagree.
    - *Argument:* "Role Separation was successful, yielding true dialectical tension, even if synthesis lacks strict structural rigor."
    - *Request:* Partial credit (Score 3 or 4) for "Judicial Nuance."

Key Integration Steps

The file below contains the complete, machine-readable JSON specification. It is designed to be loaded via json.load() and distributed into your agent's context window based on the artifact they are assigned to investigate.

To ensure the agents in your **Automaton Auditor** swarm act with precision, they use a **Targeting Protocol**. The RepoInvestigator, DocAnalyst, and VisionInspector filter the machine-readable rubric by a target_artifact key. This ensures the Code Detective doesn't attempt to "grep" a PDF, and the Document Detective doesn't look for "Pydantic models" in the report text. The key steps to follow are:

1. **The Context Builder:** Your ContextBuilder node should iterate through the dimensions array.
2. **The Dispatcher:** Send forensic_instruction to the detectives where target_artifact matches their capability. Send judicial_logic to the judges as part of their persona-specific system prompt.
3. **The Chief Justice:** Provide the synthesis_rules to the ChiefJusticeNode to ensure the final verdict respects the priority of facts over opinions.

This JSON allows you to update the "Constitution" centrally without redeploying agent code.

```json
{
  "rubric_metadata": {
    "rubric_name": "Week 2: The Automaton Auditor Self-Evaluation",
    "grading_target": "Week 2 Auditor Repository & Architectural Report",
    "version": "3.0.0"
  },
  "dimensions": [
    {
      "id": "git_forensic_analysis",
      "name": "Git Forensic Analysis",
      "target_artifact": "github_repo",
      "forensic_instruction": "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.",
      "success_pattern": "More than 3 commits showing clear progression from setup to tool engineering to graph orchestration. Atomic, step-by-step history with meaningful commit messages.",
      "failure_pattern": "Single 'init' commit or bulk upload of all code at once. No iterative development visible. Timestamps clustered within minutes."
    },
    {
      "id": "state_management_rigor",
      "name": "State Management Rigor",
      "target_artifact": "github_repo",
      "forensic_instruction": "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.",
      "success_pattern": "'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.",
      "failure_pattern": "Plain Python dicts used for state. No Pydantic models. No reducers, meaning parallel agents will overwrite each other's data."
    },
    {
      "id": "graph_orchestration",
      "name": "Graph Orchestration Architecture",
      "target_artifact": "github_repo",
      "forensic_instruction": "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.",
      "success_pattern": "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.",
      "failure_pattern": "Purely linear flow (RepoInvestigator -> DocAnalyst -> Judge -> End). No parallel branches. No synchronization node. No conditional edges for error handling."
    },
    {
      "id": "safe_tool_engineering",
      "name": "Safe Tool Engineering",
      "target_artifact": "github_repo",
      "forensic_instruction": "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.",
      "success_pattern": "All git operations run inside 'tempfile.TemporaryDirectory()'. 'subprocess.run()' used with error handling. No raw 'os.system()' calls. Authentication failures caught and reported.",
      "failure_pattern": "Raw 'os.system(\"git clone <url>\")' drops code into the live working directory. No error handling around shell commands. No input sanitization on the repo URL."
    },
    {
      "id": "structured_output_enforcement",
      "name": "Structured Output Enforcement",
      "target_artifact": "github_repo",
      "forensic_instruction": "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.",
      "success_pattern": "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.",
      "failure_pattern": "Judge nodes call LLMs with plain prompts and parse freeform text responses. No Pydantic validation on output. No retry on parse failure."
    },
    {
      "id": "judicial_nuance",
      "name": "Judicial Nuance and Dialectics",
      "target_artifact": "github_repo",
      "forensic_instruction": "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.",
      "success_pattern": "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.",
      "failure_pattern": "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."
    },
    {
      "id": "chief_justice_synthesis",
      "name": "Chief Justice Synthesis Engine",
      "target_artifact": "github_repo",
      "forensic_instruction": "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.",
      "success_pattern": "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.",
      "failure_pattern": "ChiefJustice is just another LLM prompt that averages the three judge scores. No hardcoded rules. No dissent summary. Output is console text or unstructured."
    },
    {
      "id": "theoretical_depth",
      "name": "Theoretical Depth (Documentation)",
      "target_artifact": "pdf_report",
      "forensic_instruction": "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'.",
      "success_pattern": "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.",
      "failure_pattern": "Terms appear only in the executive summary or introduction. No connection to
actual implementation. 'We used Dialectical Synthesis' with no explanation of how."
    },
    {
      "id": "report_accuracy",
      "name": "Report Accuracy (Cross-Reference)",
      "target_artifact": "pdf_report",
      "forensic_instruction": "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.",
      "success_pattern": "All file paths mentioned in the report exist in the repo. Feature claims match
code evidence. Zero hallucinated paths.",
      "failure_pattern": "Report references files that do not exist. Claims parallel execution but code
shows linear flow. Multiple hallucinated paths detected."
    },
    {
      "id": "swarm_visual",
      "name": "Architectural Diagram Analysis",
      "target_artifact": "pdf_images",
      "forensic_instruction": "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'.",
      "success_pattern": "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.",
      "failure_pattern": "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."
    }
  ],
  "synthesis_rules": {
    "security_override": "Confirmed security flaws (e.g., shell injection in git tools, raw os.system with
unsanitized inputs) cap the total score at 3, overriding any effort points from the Defense.",
    "fact_supremacy": "Forensic evidence (facts from Detectives) always overrules Judicial opinion
(interpretation from Judges). If the Defense claims 'Deep Metacognition' but the RepoInvestigator
found no supporting code, the Defense is overruled for hallucination.",
    "functionality_weight": "If the Tech Lead confirms the architecture is modular and workable, this
carries the highest weight for the 'Graph Orchestration Architecture' criterion.",
    "dissent_requirement": "The Chief Justice must summarize why the Prosecutor and Defense
disagreed in the final report. Every criterion with a score variance > 2 must include an explicit dissent
explanation.",
    "variance_re_evaluation": "If score variance across the three judges exceeds 2 for any criterion (e.g., Prosecutor says 1, Defense says 5), trigger a re-evaluation of the specific evidence cited by each judge before rendering the final score."
  }
}
```

The Tenx Evaluation Rubric

The following criteria will be used to assess your repository by the tenx grader.

Assessment Metric	**Score 1 (The Vibe Coder)**	**Score 3 (Competent Orchestrator)**	**Score 5 (Master Thinker)**
Forensic Accuracy	**Hallucination.** Agent invents files or claims code exists when it doesn't. Fails to clone the repo correctly. Output is generic text not backed by file paths.	**Basic Verification.** Agent successfully clones the repo and verifies file existence. Can find exact graph definitions (e.g., `StateGraph` instantiation) using regex or simple parsing.	**Deep AST Parsing.** Agent extracts the full commit history to verify progression. Parses the AST to confirm LangGraph logic structure (not just regex matching). Evidence is irrefutable and precise.
Judicial Nuance	**Monolithic Opinion.** Single agent acts as "The Grader." No persona separation. Scores are random or purely praise/criticism without nuance.	**Role Separation.** Distinct "Prosecutor" and "Defense" roles exist in the code. They offer different viewpoints, but the synthesis is a simple average or relies entirely on an LLM prompt.	**Dialectical Synthesis.** The Judges debate specific trade-offs (e.g., "Code is messy but innovative"). The Final Verdict is determined by deterministic rules, explicitly references the conflict, and explains *why* one side was overruled.

LangGraph Architecture	**Spaghetti Script.** Linear Python script with no state management. Hardcoded paths. No error handling.	**Functional Graph.** Nodes pass typed state correctly. Basic error handling (e.g., invalid repo URL). Judges return structured JSON via Pydantic.	**Robust Swarm.** Uses parallel execution for Detectives and Judges. Implements data reducers to prevent overwrites. State schema is strictly typed with Pydantic.
The Feedback Loop	**Ignored.** Trainee ignored peer feedback. Week 2 code remains unchanged. No self-reflection.	**Responsive.** Trainee fixed basic bugs pointed out by peers. Reflection document acknowledges the feedback.	**MinMax Optimization.** Trainee used peer agents to find deep architectural flaws in Week 2 work *AND* updated their own Week 2 agent to detect those flaws in others.
Report Quality	**Unusable.** Generic text. No file paths. No actionable advice. "Good job" or "Bad job."	**Standard.** Lists missing files and gives a score. Basic advice ("Fix the syntax error").	**Executive Grade.** Detailed "Remediation Plan" with specific instructions. Explains the "Why" (referencing Dialectical Synthesis). Professional formatting.