1. Project Vision & Core Concept

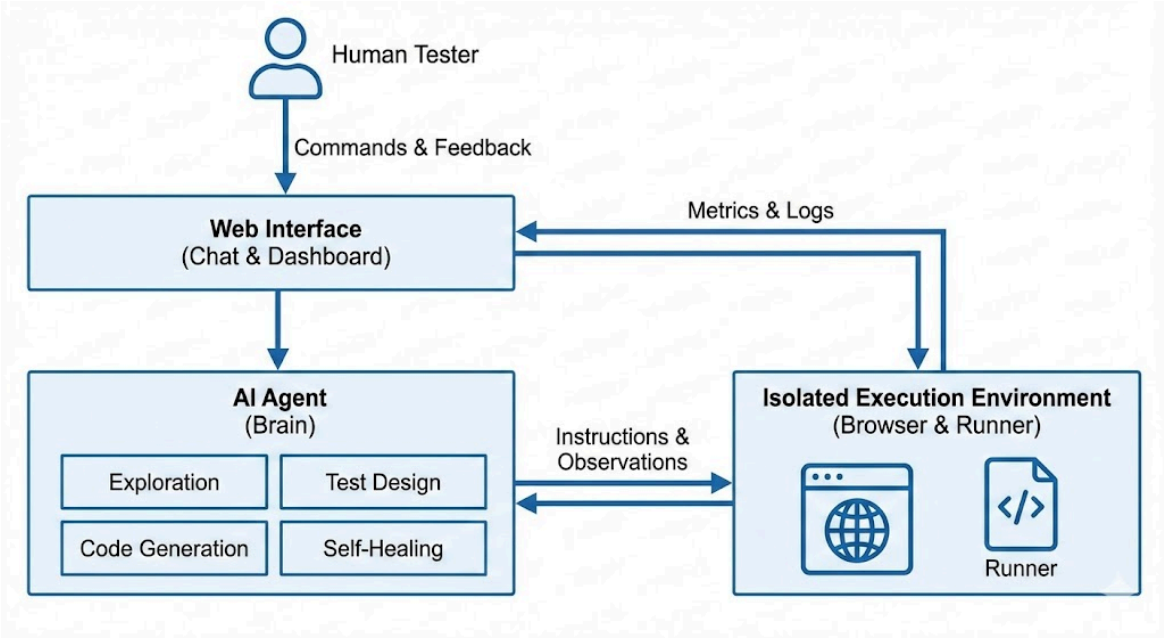
The objective is to build a **Web-based Testing Agent** that acts as a capable partner for QA Engineers. This project is not about building a generic script; it is about creating a "Human-in-the-Loop" assistant that solves real-world testing challenges: exploration, test design, implementation, and maintenance.

The core challenge is Creativity in Orchestration:

Students are tasked with designing an architecture where multiple AI capabilities (or distinct Agents) collaborate. The exact implementation details—how data flows, how state is managed, and how the UI behaves—are open to innovation, provided the system fulfills the functional goals described below in their chronological sequence.

System Architecture Goals:

- **Interface:** A web-based chat interface facilitating a dialogue between the Human Tester and the Agent.
- **Visual Context:** The Agent must control a visible browser instance. The user should see the work happening in real-time to build trust.
- **Execution Isolation:** The testing environment (the browser and runner controlled by the Agent) must run in isolation from the system's core logic to ensure stability and safety.
- **Observability & Metrics:** The system must provide insight into the "Agent's Brain."
    - Must display granular metrics: **Average Response Time (per iteration/page)** and **Tokens Consumed (per iteration/page)**.
    - Visualization using tools like MLFlow, LangFuse, or custom dashboards.
- **Safety Guardrails:** The system must implement guardrails to protect against malicious inputs and filter out instructions that are not relevant to the testing context (e.g., rejecting prompts to "write a poem" or "delete system files").
- **Parallelism:** If a Multi-Agent architecture is utilized, the system must demonstrate **Parallelism**. Agents should work concurrently where logical (e.g., one agent documenting while another explores).

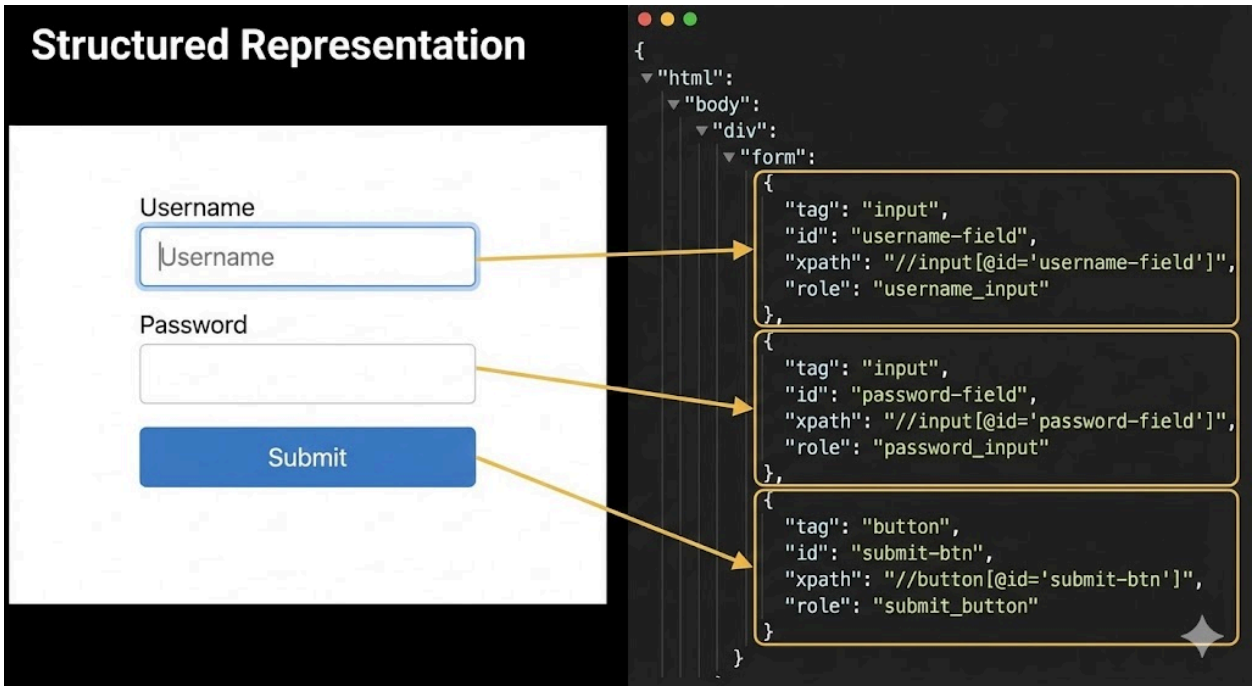


2. Functional Workflow (Chronological & Dependent)

The project follows a linear user journey. Each phase produces an output that serves as the necessary foundation for the next. **The format of these outputs is flexible, as long as they effectively enable the subsequent phase.**

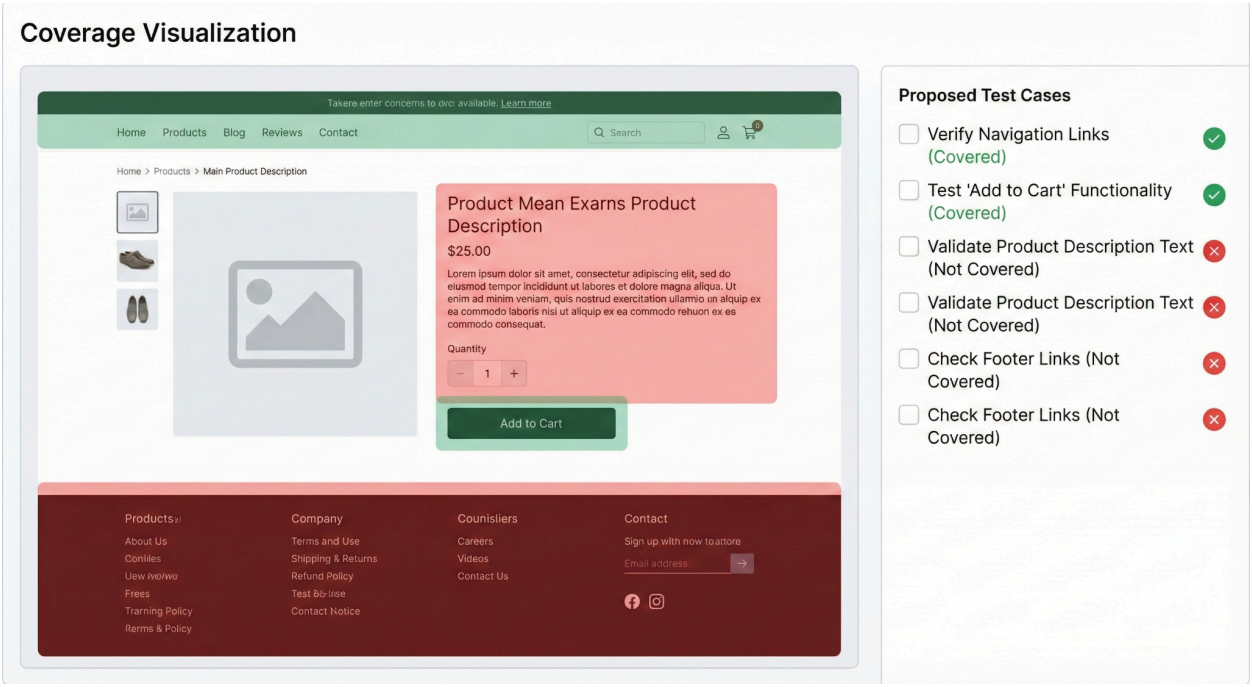
Phase 1: Exploration & Knowledge Acquisition

- **Goal:** The Agent must ingest a URL and obtain a deep understanding of the page's structure, logic, and interactivity.
- **Creative Opportunity:** How does the Agent "see"? Does it rely on the DOM? Screenshots? A hybrid approach? Does it click around randomly or follow heuristics?
- **Required Outcome:** The Agent must produce a structured representation of the page.
    - This output must be detailed enough to serve as the "ground truth" for generating test cases later.
    - It should capture element candidates (locators), potential descriptions, or visual signatures to aid in future self-healing.



Phase 2: Collaborative Test Design

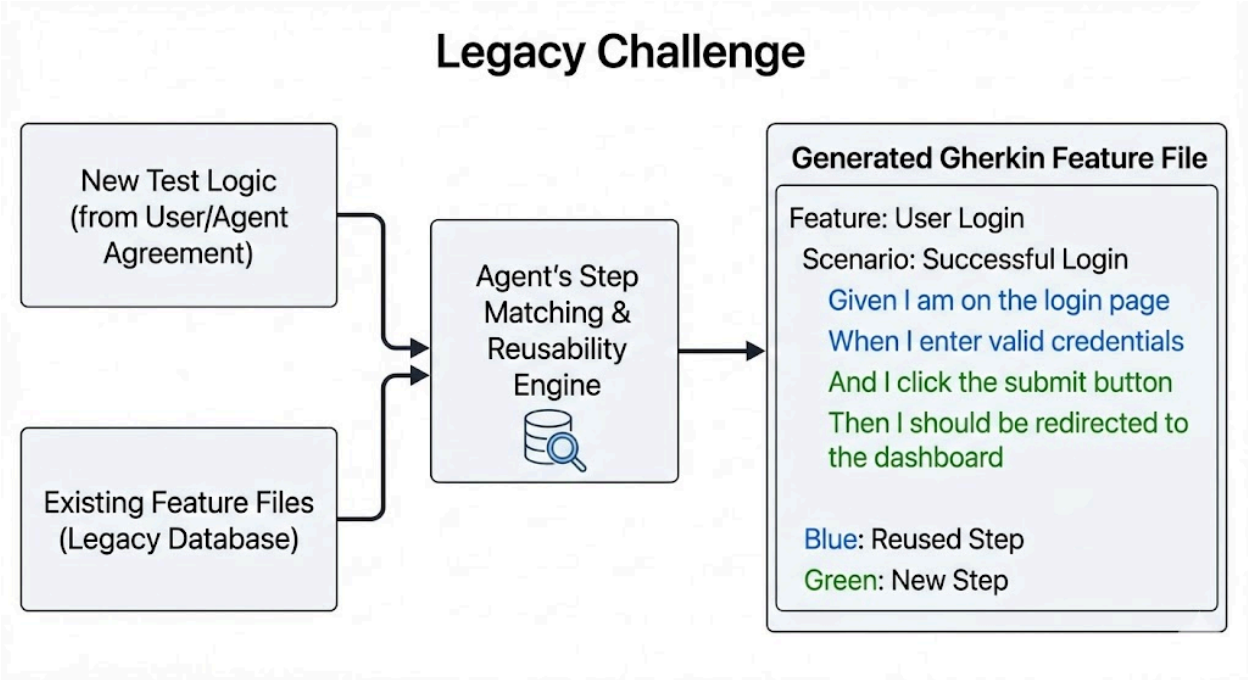
- **Goal:** To reach an agreement between the Human and the AI on *what* needs to be tested to ensure high coverage.
- **The Problem to Solve:** How do we visualize coverage?
    - The Agent should propose a list of logical test cases (e.g., a table or list).
    - **Visual Aid:** The system must provide a visual indication of coverage (e.g., taking a screenshot and "shading" or highlighting the areas covered by the proposed tests) so the user can spot gaps immediately.



- **Interaction:** The user acts as a reviewer. The Agent refines the plan based on feedback.

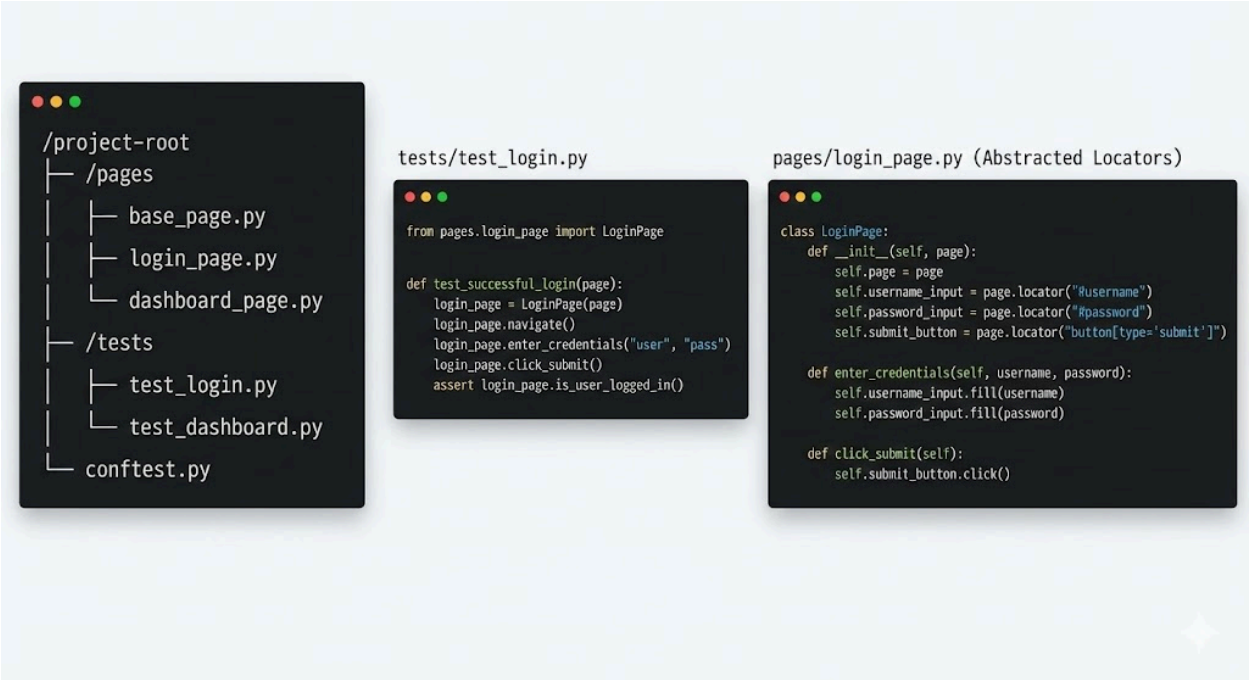
Phase 3: BDD Scenario Generation

- **Goal:** Translate the agreed-upon test logic into industry-standard behavior specifications (Gherkin/BDD).
- **Constraint - The "Legacy" Challenge:** The Agent cannot just write new code in a vacuum. It must behave like a team member joining an existing project.
    - **Context Awareness:** It must analyze existing feature files (if provided).
    - **Reusability:** If a step like Given I am on the login page already exists, the Agent *must* reuse it.
    - **Minimum Redundancy:** using outline, background...etc when needed.



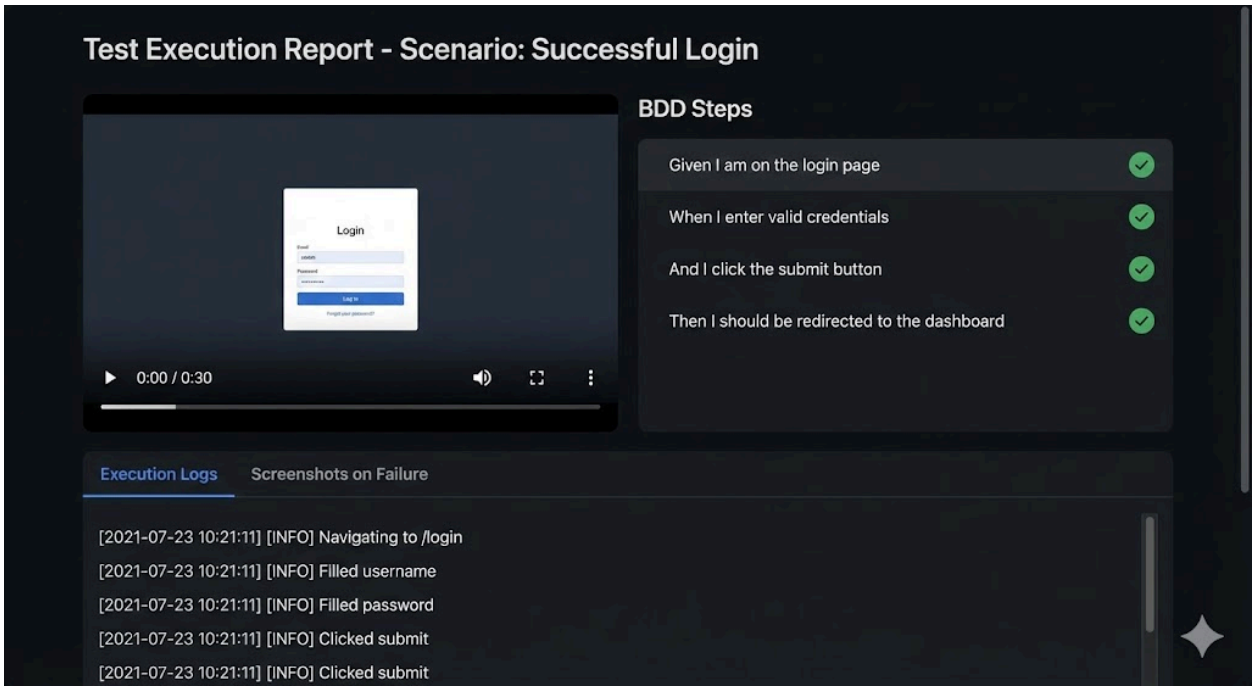
Phase 4: Implementation (Code Generation)

- **Goal:** Generate executable, clean, and maintainable test code e.g. (Playwright + Python).
- **Architecture Standard:** The output must utilize the **Page Object Model (POM)**.
- **Creative Opportunity:**
    - **Locator Strategy:** How does the Agent select the best locator? (ID vs. CSS vs. XPath vs. Semantic).
    - **Self-Correction:** How does the Agent verify the code works *while* writing it?



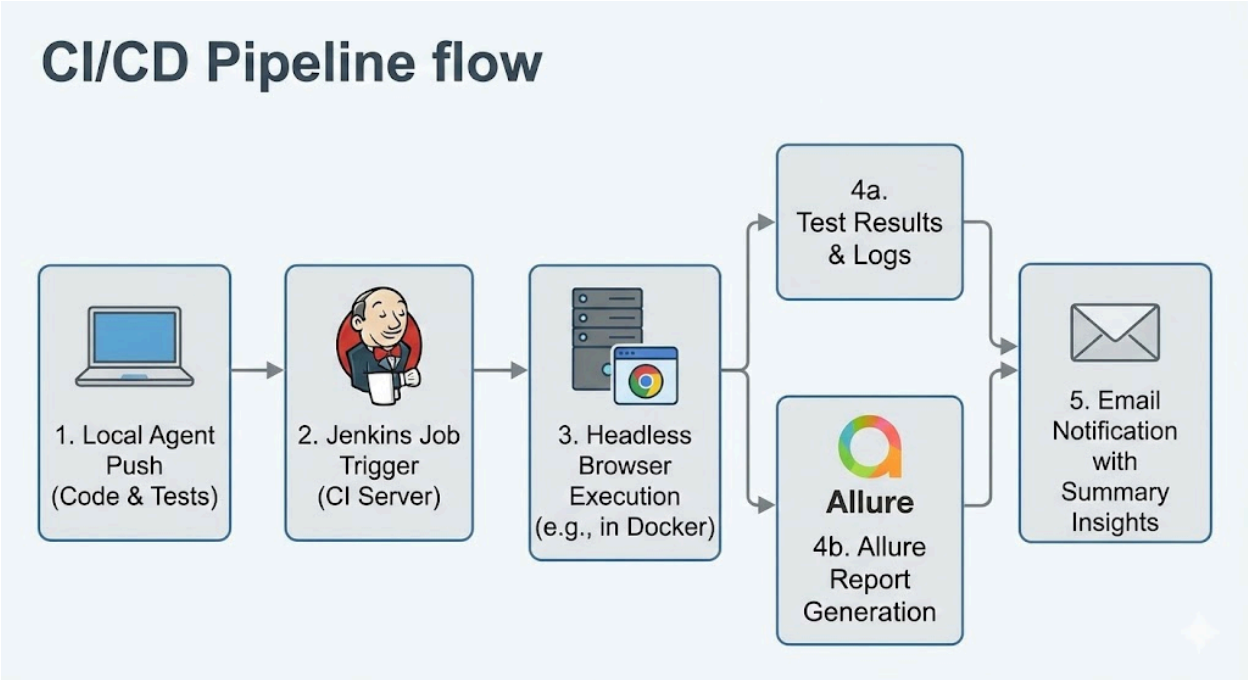
Phase 5: Verification & Trust Building

- **Goal:** Prove to the user that the tests pass and verify the expected behavior.
- **The Output:** The user needs evidence.
    - This could be a report with screenshots, a generated video of the execution, or a frame-by-frame breakdown of the BDD steps.
    - The "Review" phase is critical—the user must be able to critique the run, and the Agent must be able to refactor based on that critique.



Phase 6: Pipeline Integration (CI/CD)

- **Goal:** Transition from a local session to an automated workflow.
- **Action:** The Agent should interact with a CI tool (e.g. **Jenkins**) to create a persistent job for these tests.
- **Success Metric:** The Agent handles the complexity of configuration (scheduling, Allure report integration, custom runners).



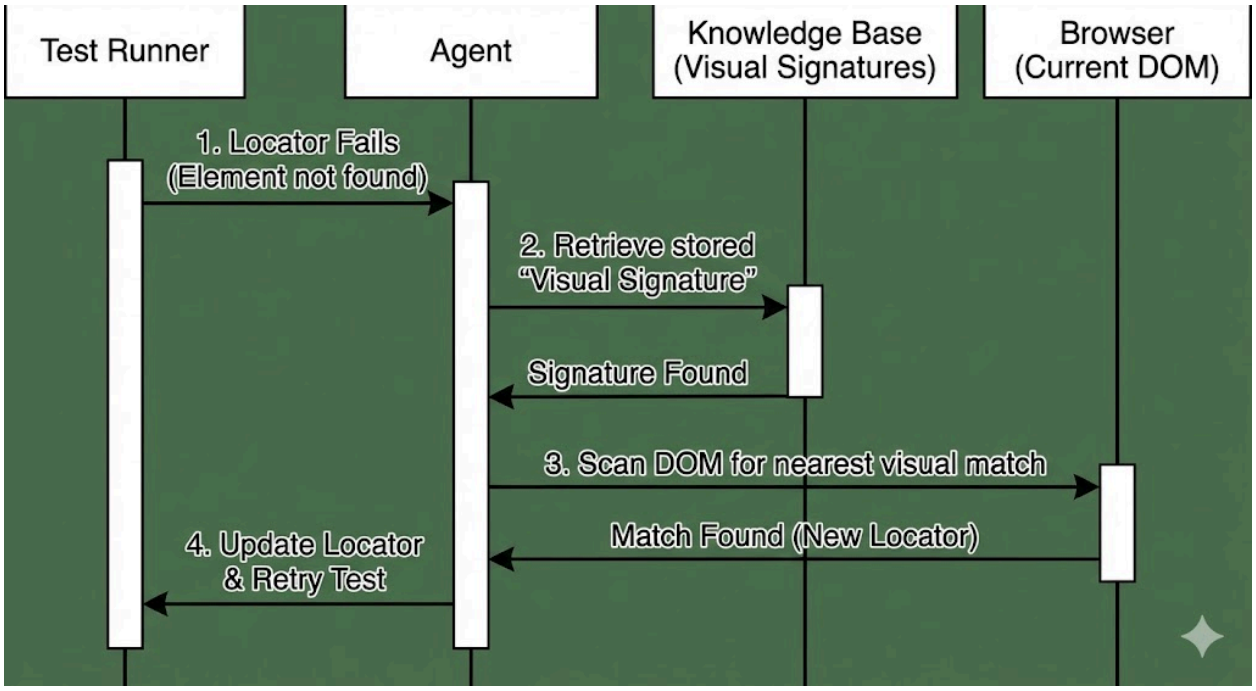
Phase 7: Continuous Monitoring & Insight

- **Goal:** Provide actionable intelligence over time as the job runs repeatedly.
- **Requirement:** The Agent should analyze historical run data (via Allure or logs) and provide a **summarized insight** (e.g., via Email).

Phase 8: Maintenance (Healing & Extension)

This is the most technically challenging and creative phase. The agent should know automatically that the page has changed and should ask if the user wants healing or extension (which means the agent remebers that he explored this page and remebers how it looked like by using his previos knowledge.)

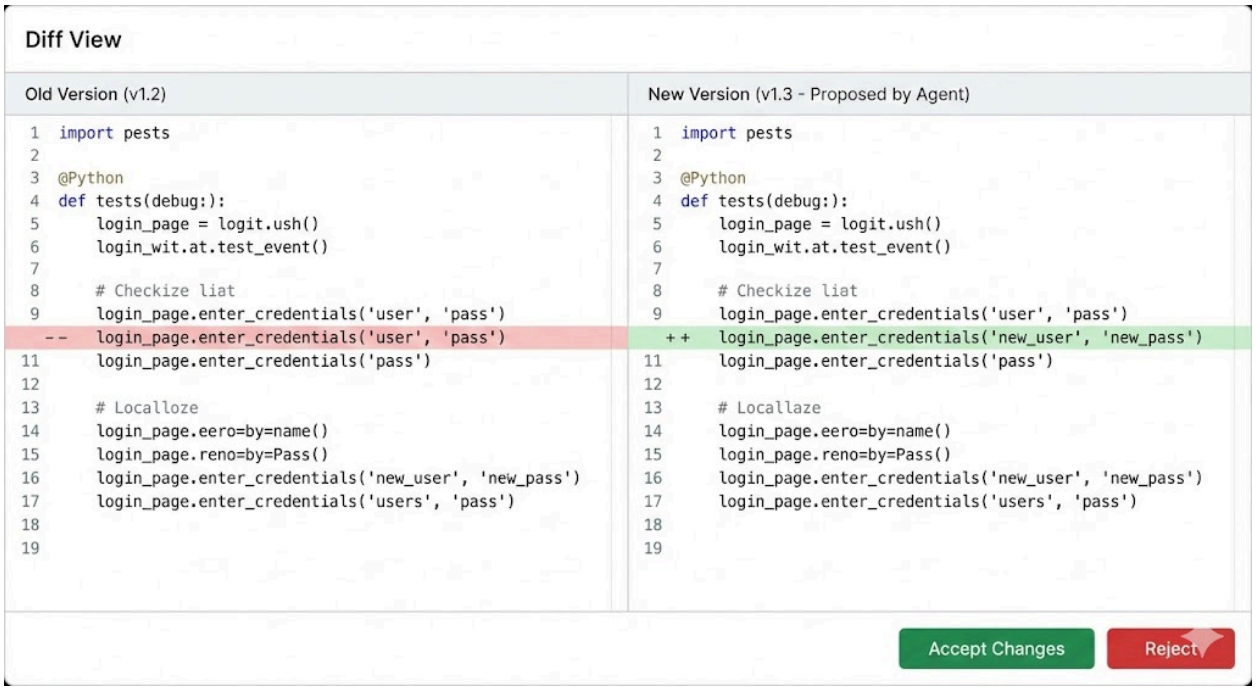
- **Sub-Feature A: Self-Healing**
  - **Scenario:** A test fails.
  - **Goal:** The Agent determines *why*. If it is a **Locator Change**, the Agent should use its previous knowledge (images/descriptions) to "heal" the implementation without rewriting the logic.



- **Sub-Feature B: Extension**
    - **Scenario:** The webpage design changes (e.g., a new field is added).
    - **Goal:** The Agent detects the visual/DOM difference, alerts the user, and updates the suite to include the new elements without breaking old tests.

3. Additional Creative Challenges & System Utilities

- **Versioning & State Management:**
    - **Diff View:** The system must clearly show the difference between the "Old Version" and the "New Version" of any artifact (Code, BDD, or Test Tables).
    - **Undo/Redo:** Users must have the ability to swap between versions or undo an Agent's action if the result is unsatisfactory.



- **"Find Bugs" Mode:** A mode where the user unleashes the Agent to perform exploratory testing and report potential edge cases.
- **Manual Assistant:** A utility where the user commands the Agent to perform repetitive tasks to assist in manual verification.
- **Context Management:** Capabilities to "Reset" the Agent to a clean slate.
- **Model Constraints:**
    - **No Paid Providers:** Teams are restricted from using paid-subscription-only LLM providers for the final delivery. The solution must utilize Free Tier APIs (e.g., Gemini Free, Hugging Face), Open Source models, or Local LLMs.

4. Evaluation Criteria

General Compliance: The evaluation will assess the successful implementation of **all** functional requirements, architectural constraints, and deliverables outlined in the document above (Sections 1, 2, and 3). If a feature is listed as a requirement (e.g., Versioning, Jenkins Integration), it is subject to grading.

The specific grading pillars are as follows:

1. **Accuracy & Truthfulness (Critical Pass/Fail):**
    - **No Hallucinations:** Zero tolerance for generating locators, steps, or elements that do not exist on the page.
    - **Healing Accuracy:** The self-healing mechanism must accurately identify the correct new element; guessing or rewriting logic to force a pass is unacceptable.
    - **Test Validity:** Generated tests must accurately verify the intended behavior.

2. **Student Competence & Code Defense:**
    - **Code Understanding:** Students must demonstrate a deep, line-by-line understanding of **both** the Agent's source code AND the test code generated by the Agent. "The AI wrote it" is not a valid defense during discussion.
    - **Model Compliance:** Verification that no paid-subscription LLM APIs were used for the final delivery.
3. **Architectural Integrity & Isolation:**
    - **System Isolation:** The Agent and its browser runner must operate in isolation from the core system logic to ensure stability.
    - **Parallelism:** If multiple agents are used, the system must demonstrate effective parallel execution.
    - **Clean Code:** Adherence to the Page Object Model (POM) in generated code and clean architecture in the Agent's codebase.
4. **Feature Completeness & Usability:**
    - **Workflow Completion:** Successful execution of all phases (Explore → Design → Implementation → CI/CD → Maintenance).
    - **Versioning & State:** Functional implementation of **Undo/Redo** capabilities and **Diff Views** (Old vs. New versions) for all artifacts.
    - **UX & Guardrails:** The system must effectively filter malicious/irrelevant instructions and provide a transparent, intuitive interface for the user.
5. **Metrics & Observability:**
    - **Visibility:** The system must clearly display **Average Response Time** and **Token Usage** (per iteration or page).
    - **Performance:** Evaluation of the execution speed for "Explore" and "Heal" phases.
    - **Coverage:** Accuracy of the visual coverage reporting (shading/screenshots).

5. Deliverables & Deadline

Deadline: Week 13

Required Deliverables:

1. **Source code** (complete and organized)
2. **Documentation** (setup, usage, architecture, instructions)
    a. Screenshots of Visualizations | interface | observability tool outputs
    b. **System design description** (how components interact with each other)
3. 1-3 demos, including:
    a. Working websites the agent has completed
    b. Chat history
    c. Test code
    d. Coverage report
    e. Logs & Traces
4. Bonus: running agent on of the benchmark