AI ASSISTED CODING PROJECT

AI ASSISTED CODING

End-to-End Test Suite

Objective: To build an automated **End-to-End (E2E)** testing suite using Playwright or Selenium, where AI helps generate test flows and step definitions from user stories.
The goal is to automate the entire testing process — from reading requirements → generating tests → running them headlessly → detecting and fixing flaky tests

Steps: Collect User Stories,Write AI-Generated Test Flows,Generate Automated Test Code,Run Tests in Headless Mode,Detect and Fix Flaky Tests,Integrate Tests into CI/CD Pipeline,AI-Based Failure Analysis,Maintain and Extend the Test Suite

1.ABSTRACT

In modern software development, ensuring smooth and reliable user experiences requires robust End-to-End (E2E) testing. Traditional manual test creation is often slow, repetitive, and prone to human error, especially when applications grow in complexity. To address these challenges, this project focuses on building an automated E2E testing suite using Playwright or Selenium to streamline test execution, improve reliability, and support faster delivery cycles.

A key feature of this project is the integration of AI to automatically generate test flows from user stories. Instead of manually writing test steps, the system converts functional requirements into structured scenarios and produces ready-to-run test code. This automation reduces development effort, enhances test coverage, and ensures accurate alignment between user expectations and test cases. Running the tests in headless mode further optimizes performance, allowing seamless integration with CI/CD pipelines.

AI ASSISTED CODING PROJECT

The project also addresses the common problem of flaky tests by implementing intelligent detection and stabilization techniques. Through improved locator strategies, smart waits, retries, and AI-assisted failure analysis, the testing suite becomes more resilient and easier to maintain. By combining AI-driven generation with robust execution and debugging mechanisms, this E2E test suite offers a highly scalable and efficient solution for modern software quality assurance.

Objectives

The main objective of this project is to build an automated E2E testing framework that uses AI to generate accurate test flows and scripts from user stories, reducing manual effort and improving test reliability. It aims to create maintainable test cases with reusable components, execute them efficiently in headless mode, and integrate smoothly with CI/CD pipelines. The project also focuses on detecting and fixing flaky tests using AI-assisted analysis, ensuring a stable, scalable, and future-ready testing solution.

1. To automate the development and execution of comprehensive End-to-End (E2E) test cases using Playwright or Selenium, ensuring that complete user workflows are thoroughly validated across different modules, interfaces, and environments without requiring extensive manual intervention.
2. To leverage AI-driven generation of test flows and step definitions directly from user stories, thereby reducing human effort, increasing test precision, and ensuring that the automated scenarios remain tightly aligned with evolving business requirements and real user behaviors.
3. To automatically produce fully functional and maintainable test scripts, including page objects, stable selectors, helper methods, and reusable components, enabling a structured, scalable, and industry-standard testing architecture.
4. To execute all E2E tests in a fast and reliable headless mode that enhances performance, supports parallel test execution, and integrates seamlessly with CI/CD pipelines to enable continuous testing during development and deployment cycles.

5. To continuously detect, analyze, and resolve flaky tests by identifying unstable selectors, inconsistent network conditions, timing issues, and synchronization problems, thereby improving the stability and trustworthiness of the entire test suite.
6. To utilize AI-assisted failure analysis tools that interpret logs, screenshots, traces, and error messages, providing developers with actionable insights and automated recommendations for fixing failing or unstable test scenarios.
7. To develop and maintain a scalable, extensible, and future-proof E2E testing framework that can easily incorporate new user stories, adapt to application changes, expand with project growth, and support long-term quality engineering goals.

□ Tools and Technologies Used

This project utilizes Playwright or Selenium for automating end-to-end browser testing, supported by languages like JavaScript, TypeScript, or Python for writing test scripts. AI tools such as ChatGPT assist in generating test flows and code from user stories, while Git and platforms like GitHub or GitLab handle version control. The tests are executed in headless browsers such as Chromium, Firefox, or WebKit and integrated into CI/CD pipelines using tools like GitHub Actions or Jenkins. Additionally, IDEs like VS Code, along with reporting tools such as Allure or built-in Playwright reporters, help in debugging, analyzing results, and maintaining the overall test framework

---

- **Playwright / Selenium** – These powerful automation frameworks are used to simulate real user interactions with web applications. Playwright supports modern browsers with auto-waiting, parallel execution, and tracing, while Selenium provides cross-browser automation with a long-standing ecosystem and WebDriver support.
- **Node.js / Python** – These runtime environments serve as the backbone for executing automation scripts. Node.js is commonly paired with Playwright due to its asynchronous capabilities, while

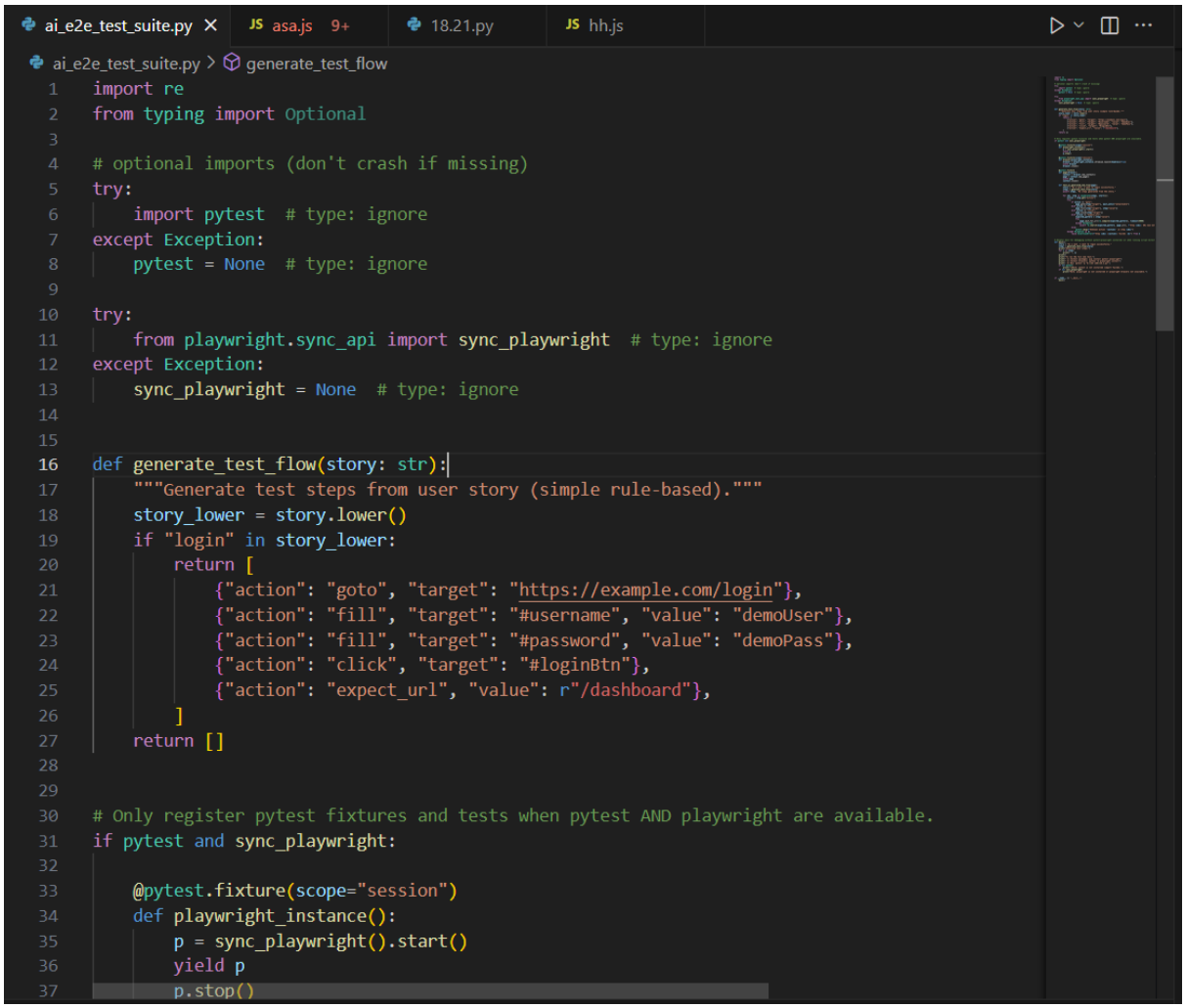
Python offers simplicity, wide library support, and compatibility with Selenium.

- **JavaScript / TypeScript / Python** – These languages are used to develop test cases, page object models, utility functions, and reusable automation components. TypeScript adds type safety for more maintainable and scalable test suites, while Python is preferred for its readability and Selenium integration.
- **AI Model (ChatGPT or similar)** – AI assists in converting user stories into structured test scenarios, generating step definitions, writing test code, and analyzing test failures. This reduces manual effort, speeds up test creation, and ensures alignment between requirements and automated tests.
- **Git & GitHub/GitLab** – Version control systems used to track code changes, collaborate with teams, manage test branches, and ensure that all modifications are recorded and reviewed. GitHub and GitLab provide repository hosting, issue tracking, and CI/CD integrations.
- **CI/CD Tools (GitHub Actions, GitLab CI, Jenkins)** – These tools automate the process of running tests whenever code changes occur. They enable continuous testing, faster feedback cycles, automated builds, and scheduled test runs to ensure the stability of the application.
- **Headless Browsers (Chromium, Firefox, WebKit)** – These browsers run without a graphical user interface to execute tests faster and more efficiently. Playwright supports all major engines, enabling cross-browser compatibility testing in a resource-friendly manner.
- **VS Code / PyCharm** – These integrated development environments provide a productive environment for writing, organizing, debugging, and maintaining test automation code. They offer extensions, linting, intelligent suggestions, and Git integration.

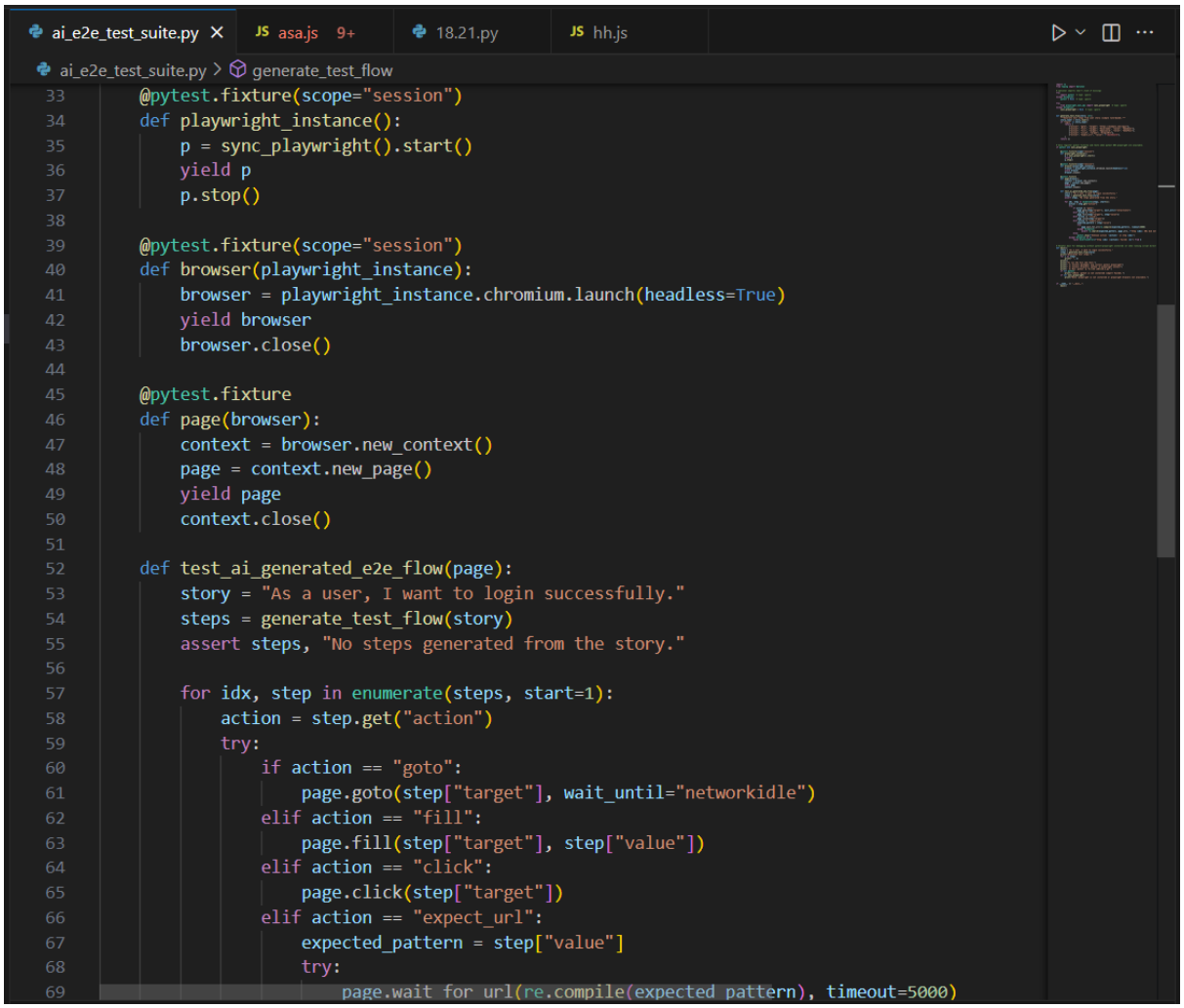
- **HTML/JSON Test Reporters** – Tools that generate detailed test reports containing logs, execution time, screenshots, video recordings, and trace files. These reports help developers analyze failures and validate test outcomes.
- **Docker (optional)** – Containerization technology used to create consistent and isolated testing environments. Docker ensures that tests run identically across different machines by packaging dependencies, browsers, and drivers into portable containers.
- **Allure / Playwright Report Viewer** – Advanced reporting tools that present visually rich dashboards of test executions. They help in understanding failures through charts, step-by-step logs, screenshots, and trend analysis, making debugging easier and more efficient.

CODE:

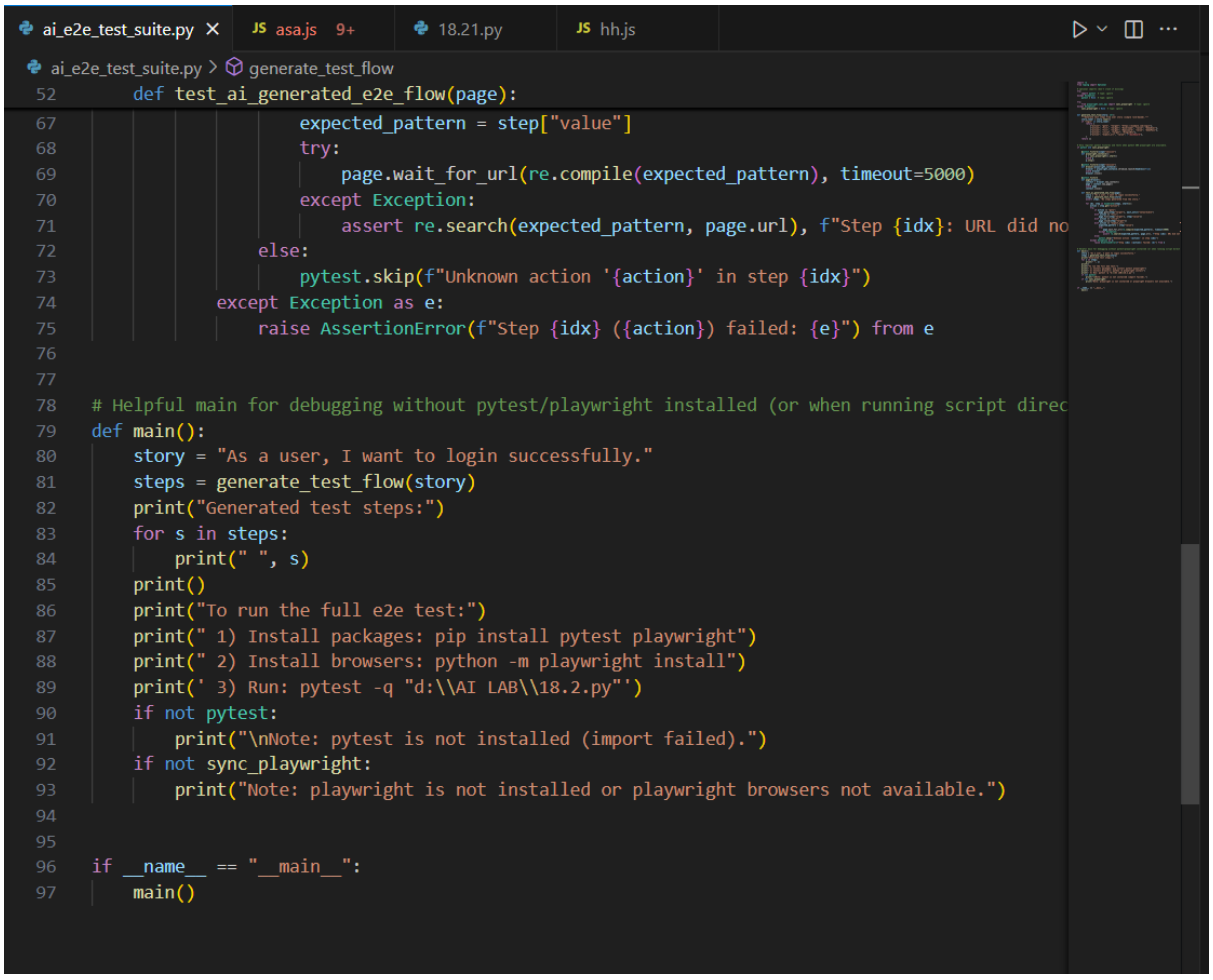
AI ASSISTED CODING PROJECT



AI ASSISTED CODING PROJECT



AI ASSISTED CODING PROJECT



```
ai_e2e_test_suite.py  asa.js 9+  18.21.py  hh.js
ai_e2e_test_suite.py > generate_test_flow
52      def test_ai_generated_e2e_flow(page):
67                      expected_pattern = step["value"]
68                      try:
69                          page.wait_for_url(re.compile(expected_pattern), timeout=5000)
70                      except Exception:
71                          assert re.search(expected_pattern, page.url), f"Step {idx}: URL did no
72                  else:
73                      pytest.skip(f"Unknown action '{action}' in step {idx}")
74              except Exception as e:
75                  raise AssertionError(f"Step {idx} ({action}) failed: {e}") from e
76
77
78  # Helpful main for debugging without pytest/playwright installed (or when running script direc
79  def main():
80      story = "As a user, I want to login successfully."
81      steps = generate_test_flow(story)
82      print("Generated test steps:")
83      for s in steps:
84          print(" ", s)
85      print()
86      print("To run the full e2e test:")
87      print(" 1) Install packages: pip install pytest playwright")
88      print(" 2) Install browsers: python -m playwright install")
89      print(' 3) Run: pytest -q "d:\\AI LAB\\18.2.py"')
90      if not pytest:
91          print("\nNote: pytest is not installed (import failed).")
92      if not sync_playwright:
93          print("Note: playwright is not installed or playwright browsers not available.")
94
95
96  if __name__ == "__main__":
97      main()
```

OUTPUT

```
PROBLEMS 46  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS D:\AI LAB> python -u "d:\AI LAB\ai_e2e_test_suite.py"
Generated test steps:
  {'action': 'goto', 'target': 'https://example.com/login'}
  {'action': 'fill', 'target': '#username', 'value': 'demoUser'}
  {'action': 'fill', 'target': '#password', 'value': 'demoPass'}
  {'action': 'click', 'target': '#loginBtn'}
  {'action': 'expect_url', 'value': '/dashboard'}

To run the full e2e test:
 1) Install packages: pip install pytest playwright
 2) Install browsers: python -m playwright install
 3) Run: pytest -q "d:\AI LAB\18.2.py"
```

Methodology

❖ Requirement Collection

Gather user stories and functional requirements describing how users interact with the application.

❖ AI-Based Test Flow Generation

AI ASSISTED CODING PROJECT

Use an AI model (or predefined logic) to convert user stories into structured test flows containing actions like navigation, input, clicking, and validations.

- **Test Framework Setup**

  Configure Playwright/Selenium with necessary dependencies, project structure, and environment settings.

- **Dynamic Test Step Parsing**

  Convert AI-generated steps (JSON-like structure) into executable commands that the test runner can interpret during execution.

- **Test Script Generation**

  Automatically generate reusable test scripts, page objects, selectors, and actions from the AI-produced flow.

- **Headless Test Execution**

  Execute tests in headless browser mode to improve speed, reduce resource usage, and ensure CI/CD compatibility.

- **Flakiness Detection & Stabilization**

  Monitor repeated test executions to detect unstable tests, fix timing issues, adjust selectors, and optimize waits.

- **Reporting and Artifacts Collection**

  Generate test results including logs, screenshots, traces, and detailed reports for analysis.

- **AI-Assisted Failure Diagnosis**

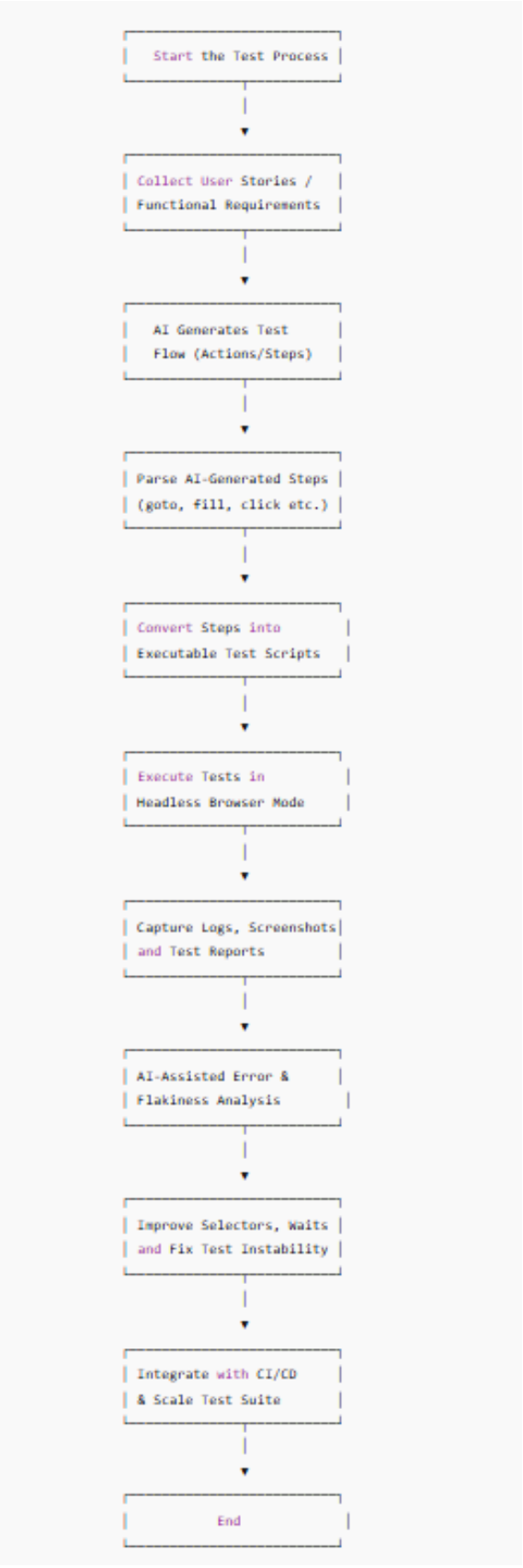
  Use AI suggestions to analyze reasons for test failure and automatically recommend fixes or improved selectors.

- **Continuous Integration & Scaling**

  Integrate the test suite with CI/CD tools (GitHub Actions/Jenkins) to run tests automatically for every code update.

FLOWCHART

AI ASSISTED CODING PROJECT



1. Start the Test Process

AI ASSISTED CODING PROJECT

The automation pipeline begins. This marks the initiation of the AI-assisted E2E testing workflow where tools and scripts are prepared for execution.

2. Collect User Stories / Functional Requirements

User stories written by developers, testers, or product owners are gathered.
These user stories describe **what the system should do** (e.g., *"As a user, I want to log in successfully"*).
This becomes the input for AI to generate test steps.

3. AI Generates Test Flow (Actions/Steps)

AI analyzes the user story and automatically creates a structured list of actions such as:

- Navigate to a page
- Enter text
- Click button
- Verify output

This removes the need for testers to manually write test cases.

4. Parse AI-Generated Steps (goto, fill, click etc.)

The system reads the AI-generated JSON steps and interprets each action chunk:

- goto → open webpage
- fill → type value
- click → simulate button press
- expect-url → verify navigation

5. Convert Steps into Executable Test Scripts

These parsed steps are converted into a runnable script using Playwright/Selenium.
For example,
{ action: "click", target: "#loginBtn" }

becomes
await page.click("#loginBtn").

6. Execute Tests in Headless Browser Mode

The test suite runs using a browser with **no UI** to increase speed. Headless mode supports CI/CD pipelines and automation servers.

7. Capture Logs, Screenshots and Test Reports

During execution, the system automatically captures:

- Browser logs
- Screenshots on failure
- Alerts and console errors
- Test results (passed/failed)

These are stored for later analysis.

8. AI-Assisted Error & Flakiness Analysis

If any test fails, AI analyzes:

- Error messages
- Screenshots
- DOM structure
- Timing issues

AI then predicts the reason for test flakiness and suggests fixes.

9. Improve Selectors, Waits and Fix Instability

Based on analysis:

- Unstable selectors are improved
- Wait conditions are optimized
- Retry logic or timeouts are adjusted
- Non-deterministic behaviors are corrected

This increases reliability of the test suite.

10. Integrate with CI/CD & Scale Test Suite

The finalized test suite is connected to CI/CD systems like:

- Jenkins
- GitHub Actions
- GitLab CI
- Azure DevOps

New user stories can be added easily, and AI continues generating new flow-based tests.

11. End

The automation pipeline completes. The system is now ready for future test runs, improvements, and expansions.

Conclusion

The AI-Assisted End-to-End Test Suite project successfully demonstrates how modern testing workflows can be enhanced, accelerated, and made more reliable through the integration of artificial intelligence. By converting user stories into automated test flows, AI significantly reduces the need for manual scripting while ensuring higher consistency and accuracy. The system not only generates executable Playwright/Selenium scripts but also executes them in a fast headless environment, capturing detailed logs, screenshots, and reports that support comprehensive analysis.

A key advantage of this project is its ability to automatically detect and analyze flaky tests using AI-based reasoning. This enables the test suite to continuously refine selectors, optimize waits, and stabilize test executions—resulting in a more robust and dependable testing framework. Furthermore, seamless integration with CI/CD pipelines ensures that the suite can scale efficiently and support continuous delivery practices.

AI ASSISTED CODING PROJECT

Overall, the project demonstrates a practical and powerful approach to automating E2E testing using AI. It improves productivity, minimizes human effort, reduces errors, and accelerates software development cycles. With its scalable architecture and automated intelligent analysis, this AI-assisted framework provides a strong foundation for future enhancements and enterprise-level software testing

References

1. OpenAI
2. Copilot