AI Agents to Test Websites Automatically Using Natural Language

1. Introduction

Software testing is an essential part of the software development lifecycle to ensure application quality and reliability. Traditional manual testing requires significant time, effort, and technical expertise. With the rapid growth of web applications, there is a strong need for intelligent and automated testing solutions.

This project, "AI Agents to Test Websites Automatically Using Natural Language", presents an AI-driven framework that allows users to provide test instructions in simple natural language. These instructions are automatically interpreted, converted into browser automation scripts, executed using Playwright, and validated using assertions. The system also generates execution reports and captures screenshots for failed test cases.

2. Problem Statement

Manual website testing suffers from the following limitations:

- Time-consuming execution
- Requires technical knowledge of automation tools
- Error-prone and inconsistent results
- Difficult to maintain test scripts

Hence, there is a need for an intelligent automation system that can:

- Understand human-readable instructions
- Automatically generate test scripts
- Execute tests reliably
- Provide detailed execution reports

3. Objective of the Project

The main objectives of this project are:

- To convert natural language test instructions into automation steps
- To automatically generate Playwright browser automation scripts
- To execute tests in headless mode
- To validate results using assertions
- To generate execution reports
- To capture screenshots for failed test cases

4. Technologies Used

The following technologies are used in this project:

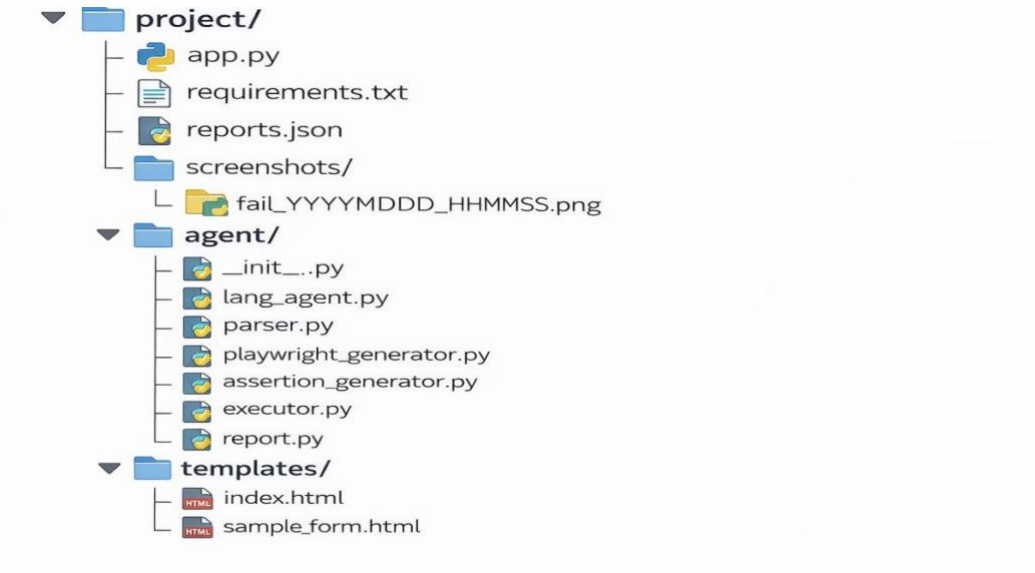
- **Python** – Core programming language
- **Flask** – Web framework for backend
- **LangGraph** – Agent-based workflow for instruction processing
- **Playwright** – Browser automation tool
- **HTML & CSS** – User interface design
- **JavaScript** – Handling frontend requests

5. System Architecture

The system follows a modular and agent-based architecture.

1. User enters a test instruction in the web interface
2. Instruction is sent to Flask backend
3. LangGraph agent processes the instruction
4. Instruction is parsed into structured test steps
5. Playwright code is automatically generated
6. Test is executed in a headless browser
7. Assertions are validated
8. Execution result is recorded
9. Screenshot is captured if test fails
10. Report is stored in JSON format

6. Project Folder Structure



7. Milestone-Wise Implementation

Milestone-1: Environment Setup and Project Initialization (Week 1–2)

In Milestone-1, the primary focus was on setting up the development environment and initializing the project structure. Python was selected as the core programming language, and essential libraries such as Flask, LangGraph, and Playwright were installed and configured. A basic Flask server was created to serve static HTML pages and handle user requests. The initial project folder structure was defined to ensure modularity and scalability. This milestone laid the foundation for the entire project by ensuring that all required tools and dependencies were correctly set up and working together.

Milestone-2: LangGraph Agent Configuration and Instruction Parsing (Week 3–4)

Milestone-2 focused on designing an intelligent agent using LangGraph to process user inputs. A baseline LangGraph agent was implemented to accept natural language instructions from the user. The input instructions were parsed using a custom parser module, which converted human-readable text into structured test steps such as opening a browser, clicking a button, or submitting a form. This milestone introduced the concept of agent-based workflows and enabled the system to understand and interpret user intentions in a structured manner.

Milestone-3: Playwright Code Generation and Automated Execution (Week 5–6)

Milestone-3 concentrated on converting parsed test steps into executable Playwright automation commands. A Playwright code generation module was developed to map parsed actions to browser automation steps. These steps were executed using Playwright in headless mode, allowing automated testing without opening a visible browser window. Assertion logic was added to validate the test results, such as checking whether a page title exists. This milestone ensured that natural language instructions could be transformed into real, executable browser tests with accurate result validation.

Milestone-4: Reporting, Failure Analysis, and Screenshot Capture (Week 7–8)

Milestone-4 enhanced the system by adding reporting and debugging capabilities. During test execution, detailed logs were generated and stored in a persistent reports.json file. Each test report includes the timestamp, user instruction, parsed steps, execution logs, and final test result. If a test case fails due to an assertion error or missing element, the system automatically captures a screenshot of the browser state and stores it in a dedicated screenshots folder. This milestone makes the system production-ready by providing clear visibility into test outcomes and simplifying failure analysis.

8. Execution Logic

PASS Case:

- All steps execute successfully
- Assertions pass
- Result marked as PASS
- Screenshot not captured

FAIL Case:

- Assertion fails or element not found
- Result marked as FAIL
- Screenshot captured automatically
- Screenshot path stored in report

9. Sample Test Inputs

- open local page and click on submit
- open youtube and click submit

10. Sample Output

PASS Output:

- Result: PASS
- Screenshot: Not required

FAIL Output:

- Result: FAIL
- Screenshot path generated

11. reports.json Explanation

The reports.json file stores execution history in structured JSON format.

Each report contains:

- Timestamp
- User instruction
- Parsed automation steps
- Execution log
- Final result
- Screenshot path (only if failed)

12. Screenshot Handling

- Screenshots are stored in the screenshots/ directory
- Screenshot is captured only when a test fails
- Timestamp-based naming avoids overwriting

13. Conclusion

This project demonstrates an AI-powered website testing automation system that converts natural language instructions into executable browser tests. By integrating LangGraph and Playwright, the system automates testing, validates results, and generates reports with failure screenshots. The approach improves reliability, efficiency, and scalability of website testing.