# Playwright + Pytest Automation with MCP-Driven Planner / Generator / Healer Workflow

---

**Version: 1.0**

**Follow 👇:**

**@akashkumarsharma03**

## ⚠️ 1. Problem We're Solving

### 1.1 Context: Playwright Test Agents are Node-centric

Playwright recently introduced Test Agents (Planner, Generator, Healer) that can explore your app, produce Markdown test plans, generate Playwright Test files, and automatically repair failing tests. However, these official agents are designed primarily for the Node.js ecosystem and the Playwright Test runner (TypeScript/JavaScript). There is no direct Python integration or API for Planner / Generator / Healer within pytest + Playwright (Python).

Organizations using Python-based automation frameworks cannot leverage these new AI-driven Playwright Test Agents out-of-the-box. They either have to:

(a) maintain a parallel TypeScript/JavaScript test suite, or

(b) manually translate generated tests to Python.

### 1.2 Objective of this Document

This document describes how to recreate a Planner → Generator → Healer workflow for a Python-based Playwright automation framework using the Model Context Protocol (MCP), GitHub Copilot, and PyCharm. The goal is to achieve similar benefits as official Playwright Test Agents while keeping the test suite in Python and following a conventional automation framework structure.

## 📚 2. Core Concepts & Key Terminology

Below are key concepts used throughout this document:

- **Playwright:** An open-source framework for browser automation that can drive Chromium, Firefox, and WebKit.
- **Pytest:** A Python testing framework that discovers, runs, and reports tests. Widely used in backend and UI automation.
- **Page Object Model (POM):** A design pattern where each page or significant component is represented by a class exposing high-level actions (e.g., login, open menu). Tests use these classes instead of raw locators.
- ● MCP (Model Context Protocol): A revolutionary protocol that enables Large Language Models to interact with external tools and systems in a structured, secure manner. MCP acts as a bridge between AI assistants and real-world applications, allowing automated workflows that were previously impossible.
- ● GitHub Copilot: An AI-powered pair programmer that integrates directly into your IDE. With MCP support, Copilot transforms from a code completion tool into an intelligent automation orchestrator capable of planning, generating, and maintaining entire test suites.

- ● Planner Agent: The strategic intelligence that explores your application's UI, understands user flows, and creates comprehensive test plans in human-readable format. Think of it as your QA architect that never sleeps.
- ● Generator Agent: The code craftsman that transforms test plans into production-ready POM-based automation code. It understands best practices, maintains consistency, and generates tests that follow your project's conventions.
- ● Healer Agent: The resilient guardian of your test suite. When UI changes break tests, the Healer automatically detects failures, re-explores the application, identifies the root cause, and proposes or applies fixes - keeping your automation healthy and maintainable with minimal human intervention.
- **MCP (Model Context Protocol):** A protocol that allows LLMs to call external tools (like Playwright, file system, or processes) in a secure and structured way.
- **GitHub Copilot:** An AI-powered coding assistant integrated into IDEs like PyCharm that can generate, refactor, and explain code. With MCP integration, it can also use tools such as Playwright MCP server.
- **Planner:** Logical AI role that explores the application and produces a human-readable test plan.
- **Generator:** Logical AI role that converts the test plan into actual POM-based automated tests.
- **Healer:** Logical AI role that analyzes failing tests, interacts with the app, and proposes or applies fixes to keep tests green as the UI changes.

## ⚙️ 3. Prerequisites & External Setup

### 3.1 Software Requirements

Ensure the following software is installed prior to starting:

This section outlines all the necessary software tools, accounts, and configurations required to successfully implement the AI-assisted Playwright automation framework. Each component plays a critical role in enabling the Planner-Generator-Healer workflow described in this document.

- Python 3.9 or higher (check via `python --version`).
- Node.js 18 or higher (check via `node -v`).
- Git (check via `git --version`).
- PyCharm (Professional Edition recommended for best Python + web support).
- A modern web browser (Chrome, Edge, etc.) for running tests via Playwright.

### 3.2 Accounts / Services

- GitHub account with an active GitHub Copilot subscription.
- Optional: any additional AI model access if used in parallel (e.g. Anthropic Claude), though this guide focuses on GitHub Copilot.

### 3.3 IDE Setup: PyCharm + GitHub Copilot

In PyCharm:

1. Open PyCharm and go to Settings → Plugins.
2. Search for "GitHub Copilot" and install the plugin.
3. Restart PyCharm when prompted.
4. Sign in to GitHub and enable GitHub Copilot when asked.

This step configures PyCharm to work with GitHub Copilot, which will serve as our AI assistant for planning, generating, and healing Playwright tests.

## 4. Creating the Automation Project (PyCharm + Playwright + Pytest)

### 4.1 Creating a New Project in PyCharm

5. In PyCharm, choose File → New Project.
6. Select "Pure Python" (or a similar template) as the project type.
7. Specify a project name, e.g. `playwright_automation`.
8. Ensure "New virtual environment" is selected so a `.venv` directory is created.
9. Click Create to initialize the project.

Why: This creates an isolated Python environment for your automation framework, preventing dependency conflicts with other projects.

### 4.2 Installing Python Dependencies

In the PyCharm terminal (within your project):

- pip install pytest pytest-playwright playwright
- playwright install

These commands install Pytest, Playwright's Python bindings, and the Pytest-Playwright plugin, and download the browsers required for automation. Always verify the environment by running a simple test afterwards.

### 4.3 Defining the Framework Structure

Create a typical automation framework layout that follows common Page Object Model conventions:

playwright_automation/

pages/ # Page Object Model (POM) classes

tests/ # Test cases using Pytest

utils/ # Shared utilities, configuration, fixtures

docs/ # Documentation and AI-generated plans

```
pytest.ini     # Pytest configuration
```

Create `pytest.ini` in the project root with:

- [pytest]
- addopts = -v
- testpaths = tests

Then add a minimal smoke test:

```
from playwright.sync_api import Page

def test_smoke_example(page: Page):
    page.goto("https://example.com")
    assert "Example" in page.title()
```

Run `pytest` to confirm everything is correctly configured. This sanity check ensures that Pytest, Playwright, and the PyCharm environment are working before adding AI and MCP on top.

## 5. Installing and Wiring the Playwright MCP Server

### 5.1 Installing the MCP Server

The Playwright MCP server acts as a bridge between the AI (GitHub Copilot) and the browser. It exposes Playwright actions (open URL, click, locate elements) as MCP tools that Copilot can call.

Install it globally via Node:

- npm install -g @executeautomation/playwright-mcp-server

### 5.2 Connecting MCP Server to GitHub Copilot

GitHub Copilot uses a configuration file to know which MCP servers it can call. You add an entry for the Playwright MCP server so Copilot can use it under the name `playwright`.

An example configuration snippet might look like:

```
{
  "mcpServers": {
    "playwright": {
      "command": "npx",
      "args": ["-y", "@executeautomation/playwright-mcp-server"]
    }
  }
```

}

After editing the MCP configuration, restart PyCharm so that the Copilot plugin reloads the list of available servers. From this point, Copilot can call the `playwright` MCP server to interact with your application via a real browser.

## 6. Defining the AI Agent Behaviour (Planner / Generator / Healer)

### 6.1 Unified Agent Prompt

To turn GitHub Copilot into a test Planner, Generator, and Healer, we define a unified prompt that explains how it should behave. Save the following into `docs/mcp_agent_prompt.md`:

You are an expert Playwright automation agent using the MCP server `playwright`.

Global rules:
- Language: Python 3, pytest, Playwright sync API
- Design pattern: Page Object Model (POM)
- Store page classes under `pages/`
- Store test files under `tests/`
- Use `from playwright.sync_api import Page, expect`
- Use the `page` fixture from pytest-playwright

You operate in three phases:

1) PLANNER
- Use the `playwright` MCP tools to open the target URL(s) and explore the UI.
- Discover pages, menus, dialogs, and critical flows.
- Produce a detailed Markdown test plan with:
  - Context (URLs, credentials, roles)
  - Scenarios (Preconditions, Steps, Expected Results)
- Save the plan as `docs/<feature_slug>_plan.md`.
- Do not generate code in this phase.

2) GENERATOR
- Read `docs/<feature_slug>_plan.md`.
- Design POM classes in `pages/`.
- Generate pytest tests in `tests/` that use those POMs.
- Use MCP tools to validate locators and timing while coding.
- Prefer semantic locators (roles, aria labels, visible text).
- Run `pytest tests/test_<feature_slug>.py --headed` if possible.

3) HEALER

- When tests fail, read the test code, POMs, and pytest error output.
- Use MCP tools in headed mode to reproduce failures.
- Identify root causes (selector changes, timing, UI updates).
- Fix POM methods and tests minimally to restore green status.
- Re-run tests until they pass.

Always:
- Preserve business intent of tests.
- Keep tests readable and maintainable.
- Explain briefly what changed during healing.

This unified prompt acts as a behavioural contract. Whenever you work with a new feature, you paste or reference this prompt in Copilot Chat so it understands how to behave as a Planner, Generator, and Healer.

## 7. Using the Planner → Generator → Healer Workflow

### 7.1 Planner Phase – Creating a Markdown Test Plan

Assume you want to automate a feature called `login_flow`. In Copilot Chat inside PyCharm:

Paste the contents of `docs/mcp_agent_prompt.md` to establish the agent behaviour.

Then instruct the agent, for example:

Feature slug: login_flow

Feature description:
- Navigate to https://your-app-url.com
- Enter valid username and password
- Submit the form
- Verify the user lands on the main dashboard

Phase to run now: PLANNER

Tasks:
- Use the `playwright` MCP server to explore this flow.
- Generate a detailed Markdown test plan.
- Save it as docs/login_flow_plan.md.
- Paste the full Markdown in your reply.

Copilot, using the Playwright MCP server, will open the browser, explore the login flow, and return a structured test plan. Save that plan as `docs/login_flow_plan.md` if it is not already created by the agent.

## 7.2 Generator Phase – Converting Plan into POM and Tests

Next, you instruct Copilot to generate POM classes and pytest tests from the plan. In Copilot Chat:

Feature slug: login_flow

We already have the plan saved at docs/login_flow_plan.md.

Phase to run now: GENERATOR

Tasks:
- Read docs/login_flow_plan.md.
- Create POM classes under `pages/` for the pages in this flow.
- Use the `playwright` MCP server to inspect the DOM and validate locators.
- Generate a pytest test file under `tests/` named `test_login_flow.py`.
- Tests must use the POMs and the `page` fixture.
- Run pytest tests/test_login_flow.py --headed if tools allow.

Return:
- The complete content of tests/test_login_flow.py.
- A brief summary of the created POM classes.

Copilot should output one or more POM files (for example `pages/login_page.py`, `pages/dashboard_page.py`) and a test file such as `tests/test_login_flow.py`. You then execute the tests from a terminal:

- pytest tests/test_login_flow.py --headed

Running in headed mode opens a visible browser so you can see exactly what the AI-generated test is doing.

## 7.3 Healer Phase – Fixing Failing Tests

If tests fail (for example due to locator changes or timing issues), the Healer phase allows Copilot to diagnose and fix problems.

Copy the failing test name and pytest error output from PyCharm.

In Copilot Chat, instruct the agent, for example:

Feature slug: login_flow

Phase to run now: HEALER

Test file:
- tests/test_login_flow.py

POM files:
- pages/login_page.py
- pages/dashboard_page.py

Failing test:
- test_valid_login

Pytest error output:
[paste full traceback here]

Tasks:
- Read the test and POM code.
- Use the `playwright` MCP server in headed mode to reproduce the failure.
- Identify incorrectly specified locators, missing waits, or changed UI.
- Update POM methods and, if necessary, the test.
- Re-run pytest tests/test_login_flow.py --headed until the test passes.

Copilot will then use Playwright MCP tools to rerun the scenario in a live browser, adjust locators or waits, and suggest code changes. You can accept and commit those changes once the tests pass.

## 8. How This Approach Solves the Original Problem

The official Playwright Test Agents (Planner, Generator, Healer) are currently tailored to the TypeScript/JavaScript Playwright Test runner, which leaves Python-based test suites without direct support. By combining GitHub Copilot, the Playwright MCP server, and a clear Planner → Generator → Healer workflow, we obtain many of the same benefits for Python:

- AI-assisted exploratory planning that generates Markdown test plans.
- Automated generation of POM-based Python tests using validated locators.
- Self-healing capabilities, where the AI diagnoses and fixes failing tests based on live browser behavior.
- Retention of an idiomatic Python project structure (`pages/`, `tests/`, `utils/`, `docs/`).

This approach allows teams invested in Python automation to adopt an AI-first testing strategy similar to Playwright Test Agents, without migrating to a TypeScript/JavaScript stack.

## 9. Sequence Diagram for Planner → Generator → Healer

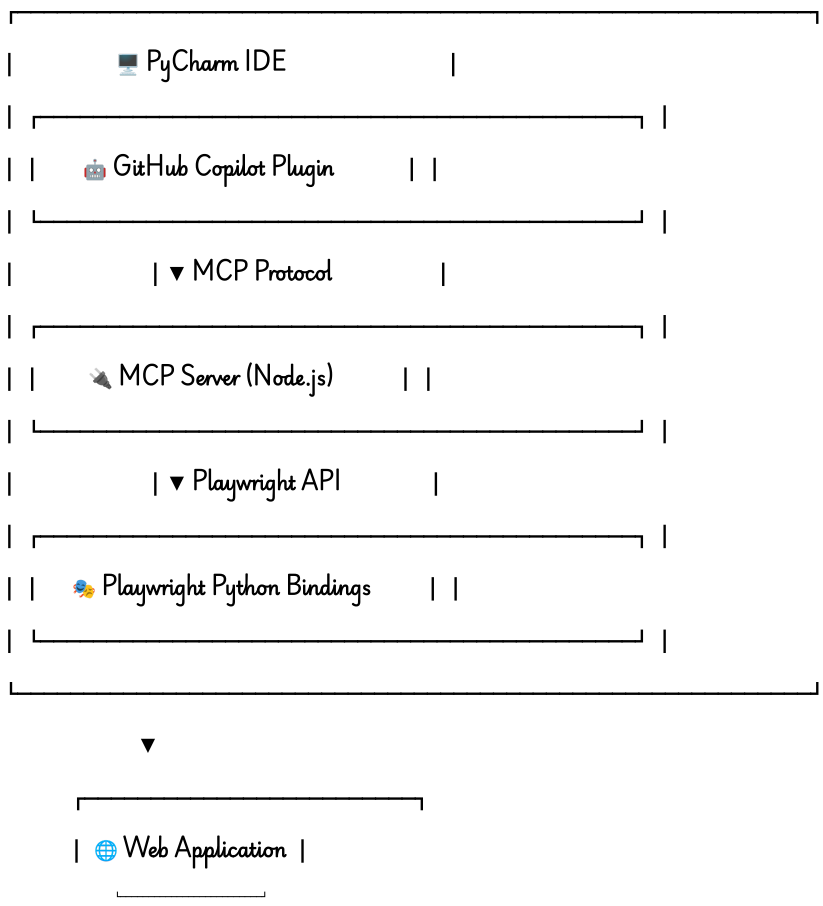
A sequence diagram can depict the interaction between Developer, Copilot, MCP Server, Browser, and Repo:
1. Planner: Developer asks Copilot to plan tests → Copilot calls MCP to explore the app →

Copilot writes Markdown plan.
2. Generator: Developer asks Copilot to generate tests from the plan → Copilot reads plan, uses MCP to validate locators, writes POM and tests, then runs pytest.
3. Healer: Developer requests healing for failing tests → Copilot reads errors and code, uses MCP to reproduce failures, updates POM and test code, then reruns pytest.

## 10. Summary

This document has outlined a complete approach to building an AI-assisted automation framework using Playwright, Pytest, GitHub Copilot, and the Playwright MCP server. The solution addresses the current limitation that Playwright Test Agents are only officially supported for TypeScript/JavaScript by recreating similar Planner, Generator, and Healer roles for Python-based frameworks. By standardizing on a familiar project structure and a unified agent prompt, teams can adopt AI-driven test planning, generation, and maintenance workflows while staying within their preferred Python ecosystem.

## 🏗️ ARCHITECTURE DIAGRAM: MCP Integration with PyCharm



## ✨ KEY BENEFITS & ADVANTAGES

By implementing this AI-assisted Playwright automation framework, teams gain significant advantages:

### 🚀 ACCELERATED DEVELOPMENT

- Reduce test creation time
- Automated test plan generation from UI exploration
- Instant POM code generation following best practices
- less manual test writing for standard scenarios

### 🔧 SELF-HEALING TESTS

- Automatic detection of UI changes
- AI-powered failure analysis and repair
- Reduced maintenance overhead
- Tests stay green despite application evolution

### 👍 PYTHON ECOSYSTEM BENEFITS

- Leverage existing Python infrastructure
- Integrate  or Enhancement with  ML pipelines in future
- Use familiar pytest framework