Playwright MCP & Agents

Automating E2E Testing with Planner and Generator Agents in a Local IDE
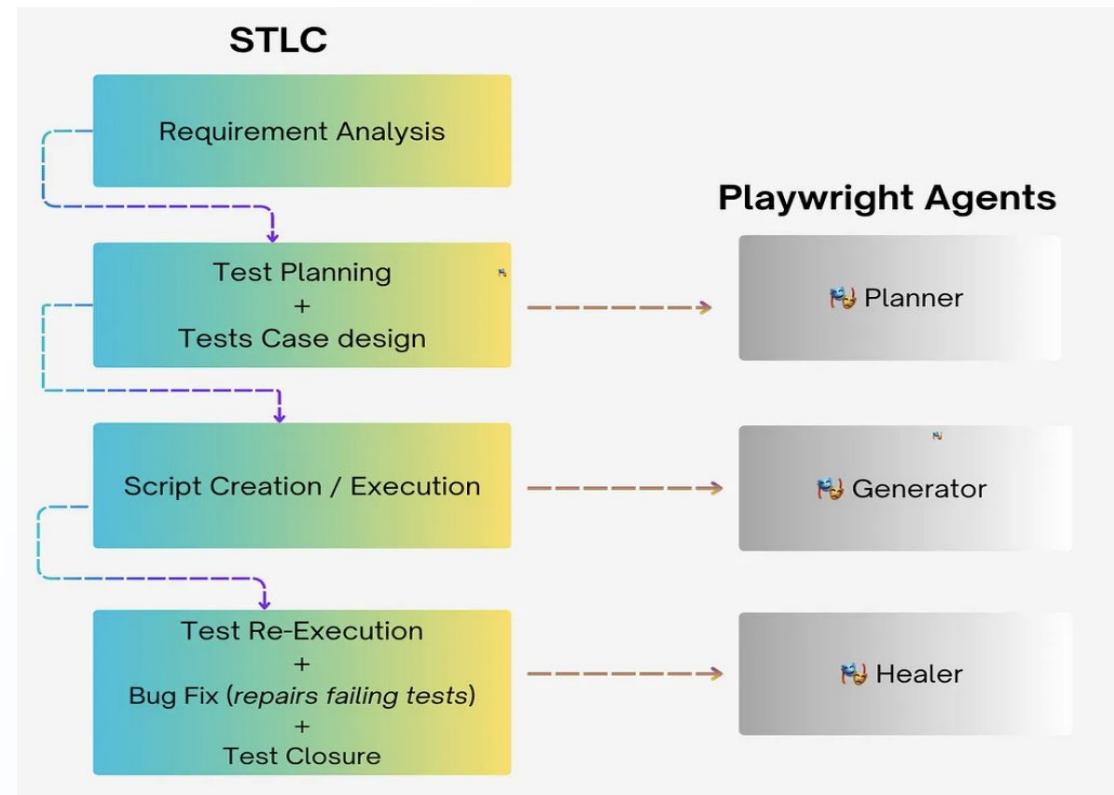
System Architecture

The Agentic Model

The system uses specialized AI agents that interact via the Model Context Protocol (MCP).

> **Planner Agent:** Explores the application and creates a high-level test plan.
> **Generator Agent:** Converts the plan into executable Playwright test code.
> **Healer Agent:** Executes the test suite and automatically repairs failing tests.
> **Local IDE:** Agents run locally, integrating with GitHub Copilot.
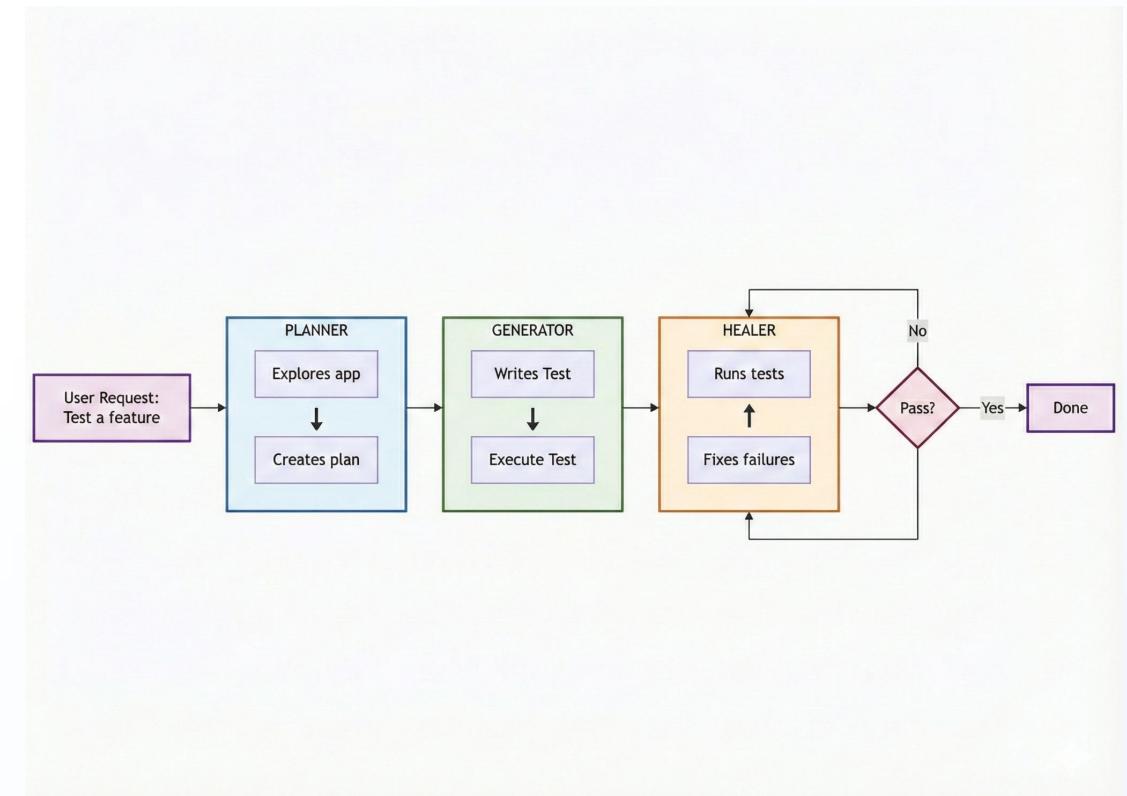
Workflow Diagram

From Prompt to Pass

This flow illustrates how a simple user intent is transformed into a verified test suite.

1. User inputs intent.
2. Planner generates `specs/*.md`.
3. Generator creates `tests/*.spec.ts`.
4. Healer fixes failing tests.
5. Playwright executes tests.
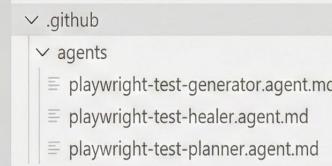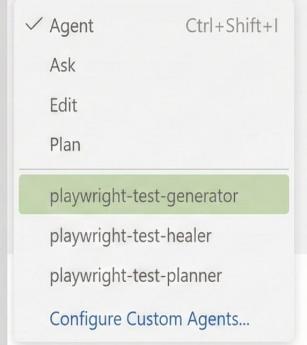
IDE Setup & Integration

MCP & Agents Configuration

We start by initializing the agents in the local environment using `npx playwright init-agents`.

The image on the right shows the agents successfully added to the IDE, ready to communicate with GitHub Copilot Chat.

Playwright Agent Configuration Workflow

Using the Planner Agent

Step 1: Initial Prompt
I engaged the Planner agent to explore the app state.

CHAT   + ⌄ ⚙ ⋯ | ⛶ ✕

← CREATING A MARKDOWN TEST PLAN FOR AN APP   ▢

                    explores the app and produces a Markdown test plan

I'll explore the task tracker application and create a comprehensive test plan. Let me start by examining the project structure and understanding what we're working with.

✓ Read project configuration and documentation files

  ● Read ⟨⟩ package.json
  ● Read ⊕ README.md
  ● Read ◇ index.html

✓ Read App.jsx and main.jsx files

  ● Read ⚙ App.jsx , lines 1 to 100
  ● Read ⚙ main.jsx

✓ Read lines 100 to 411 from App.jsx

Step 2: Strategy Generation
The Planner outputted a strategy for the test scenarios.

Perfect! I've successfully explored the TaskFlow application and created a comprehensive test plan. Here's a summary of what I discovered and documented:

Application Overview

TaskFlow is a React-based task management application with the following key features:

Core Functionality

- **Task Management**: Create, edit, delete, and mark tasks as complete/incomplete
- **Task Properties**: Title (required), description, priority (low/medium/high), due date
- **Visual Indicators**: Priority badges with color coding, completion state with strikethrough
- **Data Persistence**: Uses localStorage for data persistence across sessions

UI Features

- **Filtering**: View all tasks, active tasks, or completed tasks
- **Search**: Search across task titles and descriptions (case-insensitive)
- **Statistics**: Real-time display of total and completed task counts
- **Responsive Design**: Clean, modern UI with card-based task display

The Test Plan

Markdown Blueprint

The Planner generated this test plan (`specs/test-plan.md`).

It outlines the testing logic in plain English before any code is written, ensuring the scope is correct.

---

test-plan.md ✕

specs › test-plan.md › # TaskFlow Application Test Plan › ## Test Scenarios › ### 1. Task Management

```
1   # TaskFlow Application Test Plan
2
3   ## Application Overview
4
5   TaskFlow is a React-based task management application that allows users to create, edit, delete,
    and manage tasks with priorities, due dates, and descriptions. The application features a clean,
    modern UI with filtering capabilities, search functionality, and persistent local storage. Users
    can mark tasks as completed, filter by status (all/active/completed), and search through task
    titles and descriptions.
6
7   ## Test Scenarios
8
9   ### 1. Task Management
10
11  **Seed:** `e2e/seed.spec.ts`
12
13  #### 1.1. Create New Task
14
15  **File:** `e2e/task-management/create-task.spec.ts`
16
17  **Steps:**
18    1. Navigate to the application
19    2. Click 'New Task' button to open form
20    3. Fill in task title, priority, due date, and description
21    4. Submit the form
22    5. Verify task appears in the list
23    6. Verify task statistics are updated
```

Using the Generator Agent

Step 3: Code Generation Prompt
I instructed the Generator to implement the plan.

> transforms the Markdown plan into the Playwright Test files

I'll now transform the Markdown test plan into actual Playwright test files. Let me start by setting up the page and then generate each test file based on the plan.

✓ Ran Setup generator page – playwright-test (MCP Server)

✓ Ran Navigate to a URL – playwright-test (MCP Server)

Input

```
{
    "url": "http://localhost:5173",
    "intent": "Navigate to the application"
}
```

Output

```
### Ran Playwright code
```js
```

Step 4: Code Generation
The agent verified selectors against the live DOM and generates test cases.

Generated Test Files

I've created 6 comprehensive test files based on the test plan:

1. **Task Management Tests**

- TS create-task.spec.ts - Tests creating new tasks with all required fields
- TS edit-task.spec.ts - Tests editing existing tasks and preserving creation dates
- TS delete-task.spec.ts - Tests deleting tasks with confirmation dialogs
- TS toggle-complete.spec.ts - Tests marking tasks as complete/incomplete

2. **Filtering and Search Tests**

- TS filter-status.spec.ts - Tests filtering by active/completed/all status
- TS search.spec.ts - Tests search functionality with case-insensitive and partial matching

3. **Form Validation Tests**
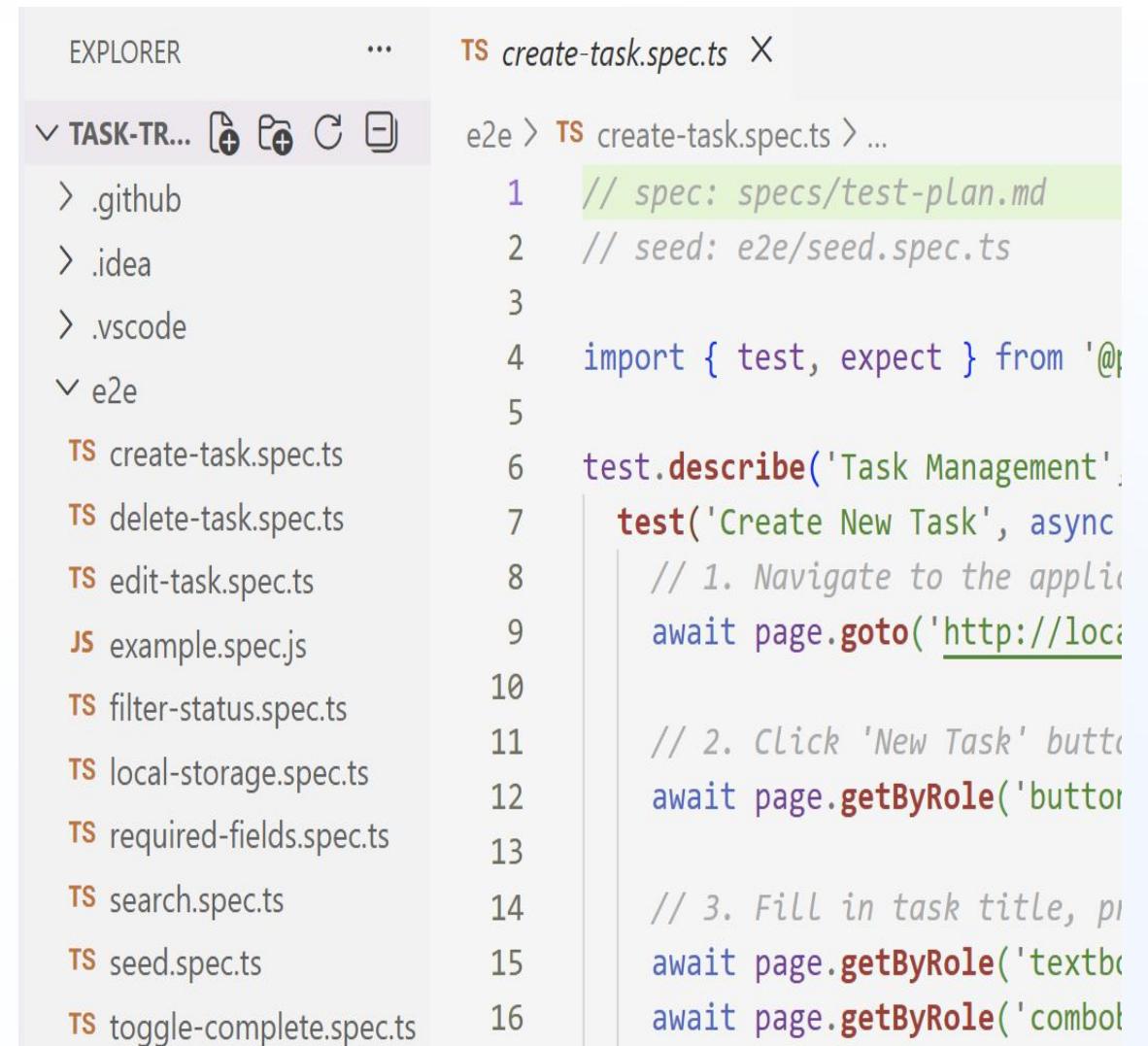
- TS required-fields.spec.ts - Tests required field validation and form behavior

Generated Spec Files

Executable Playwright Code

The Generator created this Typescript spec file
(`tests/create-task.spec.ts`).

It includes robust locators, assertions, and
follows the steps defined in the Markdown plan.

Test Results

Successful Execution

Running the generated tests yielded a passing result.

The output confirms that the agent-generated code correctly validated the application functionality.

All `33` ✓ Passed `26` ✗ Failed `7` Flaky `0` Skipped `0`

03/01/2026, 13:41:02 Total time: 30.1s

⌄ delete-task.spec.ts

✗ Task Management › Delete Task `chromium`   7.4s
delete-task.spec.ts:7

✗ Task Management › Delete Task `firefox`   10.4s
delete-task.spec.ts:7

✗ Task Management › Delete Task `webkit`   7.0s
delete-task.spec.ts:7

⌄ local-storage.spec.ts

✗ Data Persistence › Local Storage Persistence `chromium`   8.0s
local-storage.spec.ts:7

✗ Data Persistence › Local Storage Persistence `firefox`   10.7s
local-storage.spec.ts:7

✗ Data Persistence › Local Storage Persistence `webkit`   7.9s
local-storage.spec.ts:7

⌄ example.spec.js

✗ has title `chromium`   4.2s
example.spec.js:4

✓ get started link `chromium`   4.7s
example.spec.js:11

✓ has title `firefox`   2.4s
example.spec.js:4

✓ get started link `firefox`   4.0s

Future Step: The Healer Agent

Self-Healing Tests

In the future, we plan to integrate the **Healer Agent** to automatically handle broken tests.
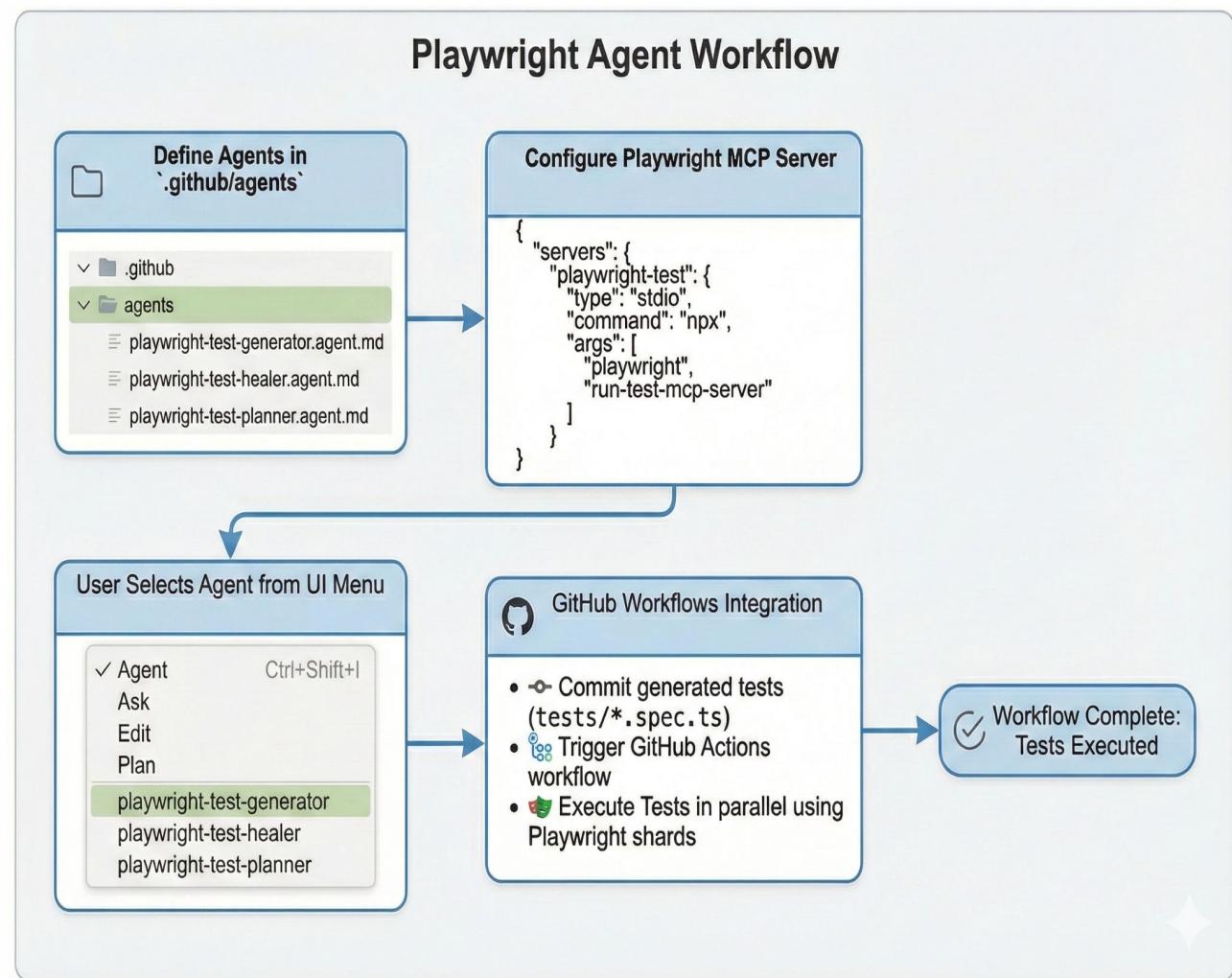
> Analyzes error traces from failed runs.

> Inspects the DOM for selector changes.

> Patches the spec file automatically to fix the break.

GitHub Workflows Integration

Automating in CI

Generated tests are standard Playwright scripts, making them fully compatible with GitHub Actions.

> **Commit:** Push generated `tests/*.spec.ts` to the repo.
> **Trigger:** GitHub Actions workflow detects changes.
> **Execute:** Tests run in parallel using Playwright shards.

Questions?

Thank you for your time.