From Brittle Scripts to Resilient Agents: The Next Evolution of UI Testing

An introduction to Agentic Testing with Playwright MCP.

We all live in the 'Maintenance Nightmare' of brittle UI tests.

Test automation promises efficiency, but reality is often a constant cycle of fixing tests broken by minor changes.

- **Flaky Tests:** Scripts fail due to timing issues or minor UI tweaks, eroding trust in the test suite.
- **High Maintenance Overhead:** QA teams spend more time updating selectors and fixing broken tests than writing new ones.
- **Coupling to Implementation:** Tests are tied to the underlying DOM structure (class names, IDs), not the user experience they're supposed to validate.

Our tests break because we tell the machine *how* to do things, not *what* we want to achieve.

The Old Way (Imperative)

We write explicit, step-by-step instructions.

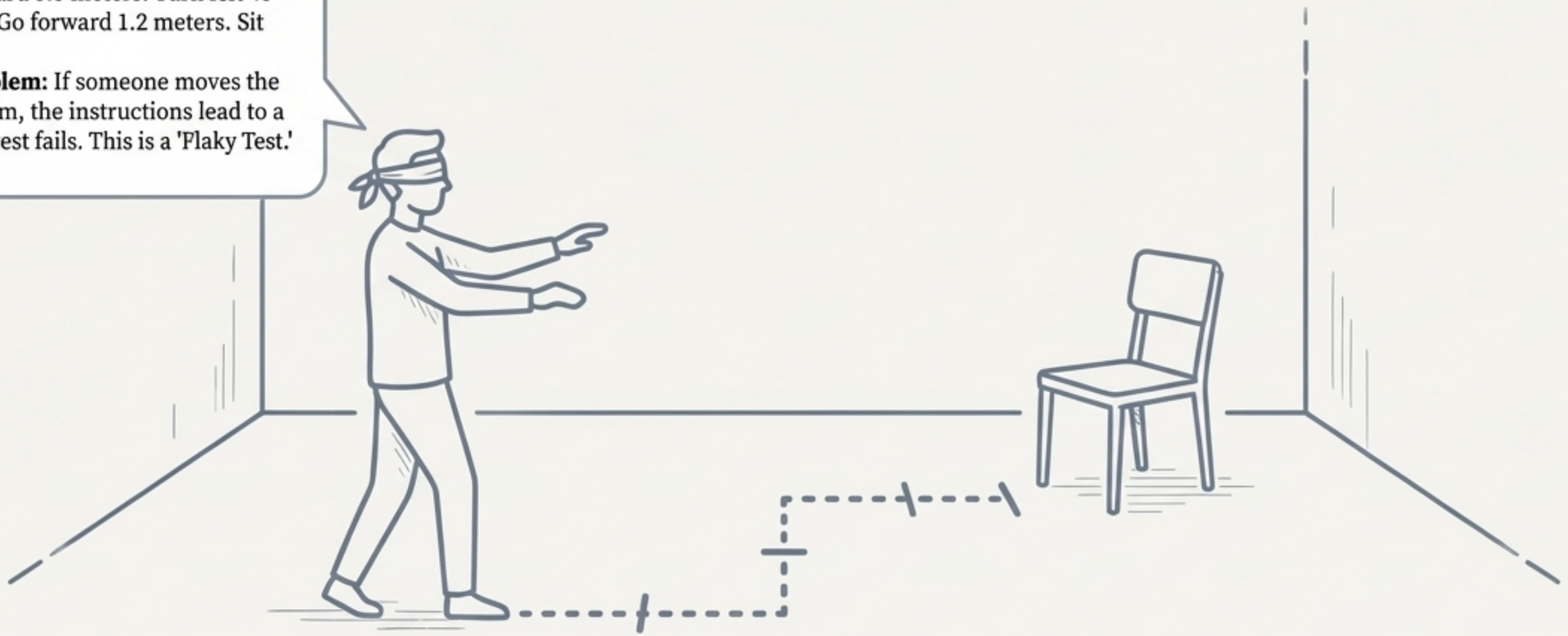
Tightly coupled to the Document Object Model (DOM).

```
page.click('#btn-submit');
page.click('.product-list > .item:first-child a');
page.click('#movie-list > div:nth-child(3)');
```

Result: If the implementation details change, the script breaks.

Traditional automation is like navigating a room in the dark with precise, fragile instructions.

The 'DOM/XPath' Method:
"Go forward 3.5 meters. Turn left 45 degrees. Go forward 1.2 meters. Sit down."
The Problem: If someone moves the chair 10cm, the instructions lead to a fall. The test fails. This is a 'Flaky Test.'



Agentic Testing shifts our focus from rigid instructions to semantic intent.

The Old Way (Imperative)

We write explicit, step-by-step instructions.

Tightly coupled to the Document Object Model (DOM).

```
page.click('#btn-submit');
page.click('.product-list > .item:first-child a');
page.click('#movie-list > div:nth-child(3)');
```

Result: If the implementation details change, the script breaks.

The New Way (Declarative)

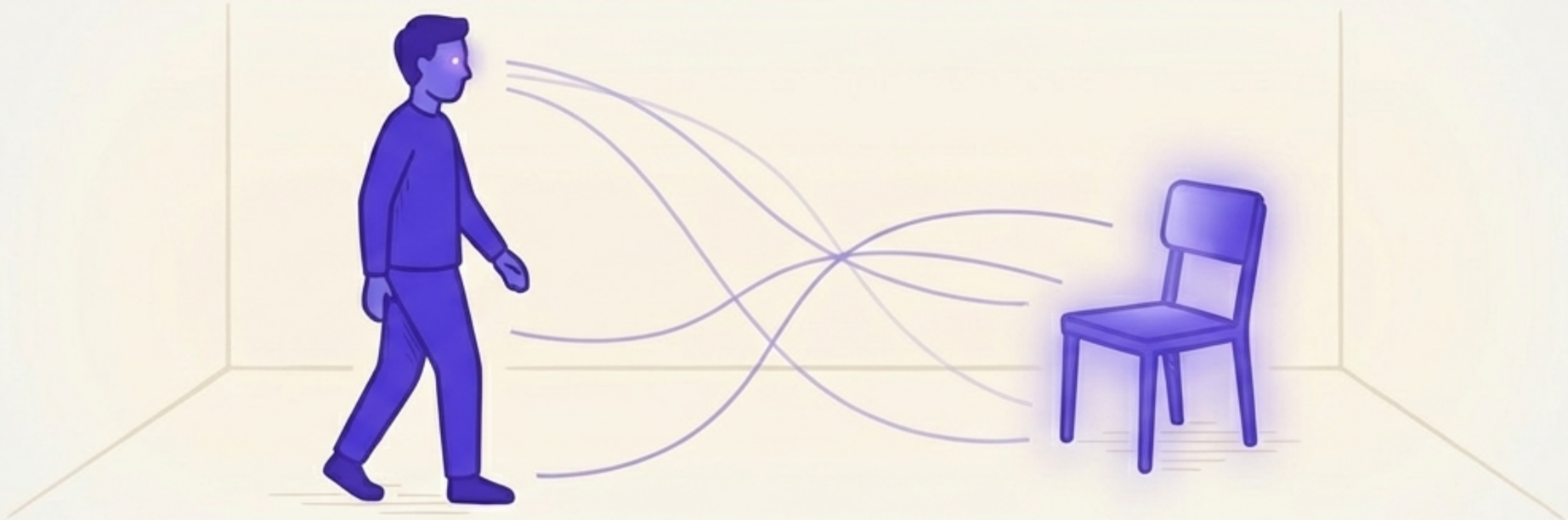
We declare the desired outcome in natural language.

Understands the page semantically, like a user.

```
Click the "Submit" button.
Click the first product in the list.
Find the movie named "Mai".
```

The agent adapts to UI changes as long as the intent is still achievable.

The new way is simply asking the agent: “Find the chair and sit down.”



“The ‘Accessibility Tree/MCP’ Method:

“Find the object that is a ‘chair’. Approach it. Sit down.”

The Advantage: The chair can be moved, restyled, or replaced, but as long as it’s semantically a ‘chair,’ the agent can find it. The test passes. This is a **Resilient Test.**

This shift is powered by understanding a page's *meaning* (A11y Tree) over its technical structure (DOM).

DOM (Document Object Model)

A detailed technical blueprint for the browser. Full of `divs`, `spans`, and complex class names like `wrapper-v2-flex`.

```
<div class="header-container-v2-flex">
  <div class="nav-wrapper-main">
    <a href="/" class="logo-link">...</a>
    <div class="search-bar-outer">
      <span class="search-icon"></span>
      <input id="q-11b" name="query" ... />
    </div>
    ...
  </div>
</div>
```

Accessibility (A11y) Tree

A semantic summary for screen readers (and AI). It filters out the noise, leaving only meaningful elements.

```
searchbox "Search store"
link "Home"
button "Sign In"
```

The A11y tree provides the context an AI needs to understand the page like a human.

An LLM is a powerful “brain in a jar”—it needs a a way to interact with the world.

The Model Context Protocol (MCP) acts as the universal connector, giving the AI “hands and eyes.”

MCP is an open standard that lets LLMs connect to external tools in a uniform way. It solves two key limitations:

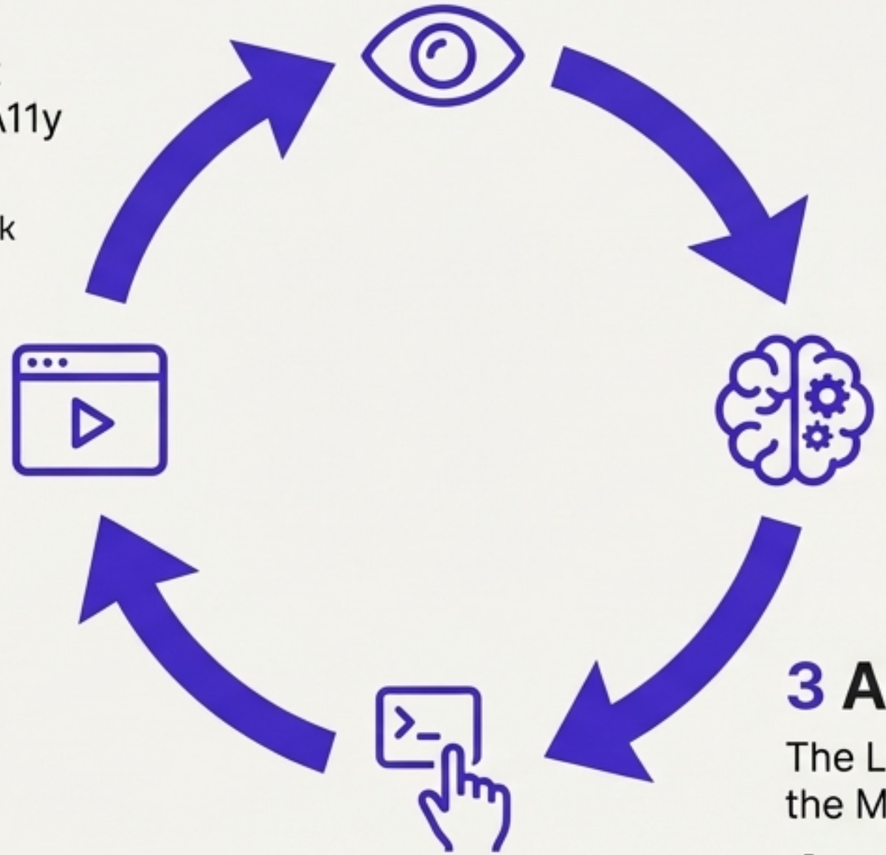
1. **Gives it a Body (Function Calling):** Allows the LLM to request actions like `click_button` or `fill_input`.
2. **Gives it Knowledge (Context Injection):** Allows tools to feed the LLM real-time data, like the A11y Tree of a live webpage.

The agent uses MCP to observe, reason, and act in a continuous loop.

1 Observe

MCP Server sends the current state of the web page (as an A11y Tree) to the LLM.

```
searchbox "Search store", link "Home"
```



2 Reason

The LLM analyzes the observation and its goal, then decides on the next action.

I see a search box. I need to type 'Laptop' into it.

3 Act

The LLM requests an action from the MCP Server.

```
playwright_fill(selector="searchbox", value="Laptop")
```

4 Execute

Playwright MCP executes the command in the browser. The loop repeats with a new observation.

The difference is stark: from writing brittle code to expressing clear, human-readable intent.

Test Scenario: Search for ‘Laptop’ and verify the ‘Add to Cart’ button appears.

Traditional Playwright Script

```
// test.spec.ts
await page.fill('input[name="q"]', 'Laptop');
await page.press('input[name="q"]', 'Enter');

// Risk: Complex class, easily changed.
await page.click('.product-list > .item:first-
child a');
await expect(page.locator('#add-to-cart-btn'))
).toBeVisible();
```

Agentic Testing Prompt

Act as a real user. Go to demo-store.com.

Search for the keyword ‘Laptop’.

Click the first result that seems most reasonable.

Finally, confirm for me that the product detail page has an ‘Add to Cart’ button.

You don't need a full-blown agent to get value. Start by by generating perfect test scripts *_today_*.

Before you replace your test suite, use Playwright MCP to accelerate writing it. This is the killer feature that provides immediate ROI.

Reduce the time spent writing boilerplate test code by up to 80%.

MCP provides the AI with perfect context, allowing it to write best-practice Playwright code for you.

1. Observe. You ask the MCP-enabled AI to look at a page (e.g., `my-app.com/login`). The MCP server opens the page and sends the clean A11y Tree to the LLM.

2. Request. You provide a simple prompt: "Based on what you see, write a complete `login.spec.ts` file to test a successful login."

3. Generate. The LLM, now armed with precise context, generates a script that uses robust, best-practice locators.

Key Insight: This is superior to just pasting HTML into an LLM because the A11y Tree context is precise and semantic, leading to better code.

The generated code is clean, robust, and follows Playwright's own best practices

```
// Code generated by an LLM with MCP context
import { test, expect } from '@playwright/test';

test('User can login successfully', async ({ page }) => {
  await page.goto('https://my-app.com/login');

  await expect(page.getByRole('heading', { name: 'Welcome' })).toBeVisible();

  await page.getByLabel('Email').fill('user@example.com');
  await page.getByLabel('Password').fill('password123');

  await page.getByRole('button', { name: 'Sign In' }).click();

  await expect(page).toHaveURL(/dashboard/);
});
```

Understands form semantics, just like a user.

Prefers robust role locators over fragile CSS/XPath.

No more manual 'Inspect Element' needed.

Agentic testing isn’t an all-or-nothing switch; it’s a spectrum of capabilities.

Phase 1: Now - Augmentation

Use Case:
Test Script Generation

Goal:
Accelerate test creation and enforce best practices. Immediately reduce manual effort.

Phase 2: Next - Exploration

Use Case:
AI-Driven Exploratory Testing

Use agents to run unscripted scenarios (e.g., scenarios (e.g., "Try to order without an address") to find bugs that rigid scripts miss.

Phase 3: Future - Autonomy

Use Case:
Self-Healing Test Suites

Goal:
Integrate agents into CI/CD pipelines where they can intelligently adapt to minor UI changes without human intervention.

The future of testing is not about writing better scripts. It’s about better expressing intent.

We are moving from a world where we test the **implementation** to one where we validate the **user experience.**

FROM	TO
Imperative (Telling the machine *how* to do something)	Declarative (Telling the machine *what* you want to achieve)