AiBuildPath
CAREER GROWTH

2026 EDITION

COMPLETE EDITION · 14 CHAPTERS + 3 APPENDICES

# The Playwright + AI Testing Playbook

Build, fix and scale Playwright tests with Claude, AI agents and MCP

- ✓ Playwright MCP & built-in test agents
- ✓ User story → plan → reviewed tests
- ✓ Fix flaky tests, run CI, ace interviews
- ✓ Free & open-source tools first

```
$ npx playwright test
✓ checkout › user completes an order
✓ cart › badge shows item count
✓ api › create and read a booking
reviewed by a human · committed
```

# Contents

AIBUILDPATH ACADEMY · The Playwright + AI Testing Playbook (2026 Edition)

CHAPTER 1 · Why Playwright + AI Changes QA Work in 2026

CHAPTER 2 · Setup: Node.js, Playwright, VS Code, AI Assistant and Playwright MCP

CHAPTER 3 · How MCP Works: Snapshots, the Accessibility Tree and Tools

CHAPTER 4 · Prompt Engineering for Testers

CHAPTER 5 · From User Story to Test Plan to Playwright Tests

CHAPTER 6 · Smart Locators and AI-Assisted Fixing of Broken Tests

CHAPTER 7 · Framework Design: Page Objects, Fixtures and Config

CHAPTER 8 · API Testing and Network Mocking with Playwright

CHAPTER 9 · Debugging Flaky Tests with Trace Viewer and AI

CHAPTER 10 · Playwright Test Agents: Planner, Generator, Healer

CHAPTER 11 · CI/CD: GitHub Actions, Reports and AI Failure Triage

CHAPTER 12 · Guardrails: Code Review, Security, Data Privacy and Cost

CHAPTER 13 · Capstone: A Complete AI-Assisted Test Suite

CHAPTER 14 · Interview Prep: 40 Questions and a 30-Day Plan

APPENDIX A · Prompt Library

APPENDIX B · Playwright + MCP Cheat Sheet

APPENDIX C · Troubleshooting Guide

AIBUILDPATH ACADEMY

# The Playwright + AI Testing Playbook (2026 Edition)

*Build, fix and scale Playwright tests with Claude, AI agents and MCP, using free and open-source tools wherever possible.*

© 2026 AiBuildPath Academy. All rights reserved. No part of this book may be copied, resold or shared publicly without written permission. Sample code in the book is yours to use in your own projects.

Playwright is a Microsoft project. Claude and Claude Code are Anthropic products. All other product names belong to their owners. This book is independent and not endorsed by any of them. Tools and prices change often, so where the book says “check the docs”, please do.

## How to use this book

This is a **lab-first** book. Every chapter teaches one skill, you build it with your own hands, and then you practise explaining it the way you would in an interview. You need about **30 days at 1 to 1.5 hours a day** (Chapter 14 has the day-by-day plan).

Every chapter follows the same shape, so you always know where you are:

| Section | What it gives you |
|---|---|
| Promise and *What you will learn* | Why the chapter matters |
| Concept | The idea in plain English, with a real QA-project example |
| Hands-on lab | Numbered steps, full code, and the output you should see |
| Prompt templates | Copy-ready prompts with placeholders like `[URL]` |
| Common mistakes / Pro tips | What goes wrong on real projects, and what experienced people do |
| Interview questions | 5 questions with strong model answers |
| Checklist | Tick it off before you move on |

### Box guide

PROMPT TEMPLATE

**Green boxes** are prompt templates. Replace anything in `[SQUARE BRACKETS]` before you send it.

FREE & OPEN-SOURCE ROUTE

Green “free” boxes show the free or open-source way to do the same thing.

**COST CHECK**

Amber “cost check” boxes tell you when a step needs a paid plan or uses paid API tokens.

**WATCH OUT**

Red boxes are mistakes that cost real teams time. Read them twice.

## Who this book is for

- QA engineers, automation testers and SDETs with 2 to 15 years of experience
- People who already know basic testing (test cases, bugs, regression) and maybe some Selenium or Cypress
- People who are new to AI-assisted testing and want a practical, honest path to using it at work

You do **not** need to be a strong programmer. If you can read a simple `if` statement and a function, you can finish this book. Every new term is explained the first time it appears.

## Tools: what is free and what is paid

| Tool | Cost | Licence | Used for |
|---|---|---|---|
| Node.js (LTS) | Free | Open source (MIT) | Runs Playwright |
| Playwright + `@playwright/test` | Free | Open source (Apache-2.0) | Browser and API testing |
| Playwright MCP ( `@playwright/mcp` ) | Free | Open source (Apache-2.0) | Lets an AI assistant drive a browser |
| VS Code + Playwright Test extension | Free | VS Code is built on open-source Code-OSS; VSCodium is a fully open-source build | Editor, running and debugging tests |
| Git + GitHub | Free | Git is open source (GPL-2.0) | Version control |
| GitHub Actions | Free for public repos; private repos get a monthly free allowance | n/a | Continuous integration (CI) |
| OpenCode | Free | Open source (MIT) | Terminal AI coding agent; you choose the model |
| Cline | Free | Open source (Apache-2.0) | AI coding agent inside VS Code; you choose the model |
| Ollama | Free | Open source (MIT) | Runs AI models on your own computer |

| Tool | Cost | Licence | Used for |
|---|---|---|---|
| GitHub Copilot | Free tier with monthly limits; paid plans | Proprietary | AI assistant in VS Code |
| Claude Code | **Paid**: a paid Claude plan (such as Pro) or Anthropic API credits | Proprietary | Terminal AI coding agent used in most examples |

**COST CHECK**

The AI **model** is the only part that usually costs money. Open-source agents like OpenCode and Cline are free, but they still need a model. A cloud model costs per use (tokens). A local model through Ollama is free but needs a decent computer, and it writes weaker tests than the top cloud models. This book shows Claude Code in most examples because it is a strong, widely used option, but **every prompt in this book works in any MCP-capable assistant**. Check current pricing on each vendor's website before you buy.

## Version policy

This book was written and type-checked against **Playwright 1.63**; 1.64 was the newest release at the time of publishing. Playwright ships a new minor version roughly every 1 to 2 months and rarely breaks the APIs used here. Check your version with:

```
npx playwright --version
```

If anything in your version behaves differently, the official docs at **playwright.dev** win. Playwright 1.62 and later need **Node.js 20 or newer**. Use the current Node LTS (22 or 24).

## The one project you will build

Every chapter adds to **one** repository called `aibuildpath-playwright-ai`. By Chapter 13 it is a complete, portfolio-ready test suite. This is the final folder layout:

```
aibuildpath-playwright-ai/
├── playwright.config.ts        # projects, reporters, timeouts (Ch 2, 7, 8, 13)
├── package.json / tsconfig.json
├── .mcp.json                   # MCP servers for Claude Code (Ch 2, 10, 12)
├── .vscode/mcp.json            # MCP servers for VS Code / Copilot (Ch 2)
├── CLAUDE.md                   # rules the AI assistant must follow (Ch 4)
├── .env.example                # copy to .env; never commit .env (Ch 7, 12)
├── specs/                      # Markdown test plans (Ch 5, 10)
├── prompts/                    # saved prompt templates (Ch 4, 11)
├── pages/                      # Page Objects (Ch 7)
├── fixtures/                   # custom fixtures (Ch 7, 8)
├── test-data/                  # users, customers, bookings, env (Ch 7, 8)
├── tests/
```

```
│   ├── setup/                  # login once, save session (Ch 7)
│   ├── ui/todo/                # TodoMVC practice tests (Ch 1, 2, 9)
│   ├── ui/shop/                # Sauce Demo e-commerce tests (Ch 2 to 13)
│   └── api/                    # API and network-mocking tests (Ch 8)
└── .github/
    ├── workflows/playwright.yml        # CI pipeline (Ch 11, 13)
    └── pull_request_template.md        # AI code review checklist (Ch 12)
```

## Practice sites (all free and public)

| Site | What it is | Used in |
|---|---|---|
| `https://demo.playwright.dev/todomvc/` | Playwright's own TodoMVC demo app | Ch 1, 2, 9 |
| `https://www.saucedemo.com` | "Swag Labs", a demo shop made for test practice | Ch 2 to 13 |
| `https://restful-booker.herokuapp.com` | A public practice API for hotel bookings | Ch 8, 13 |

> **NOTE**
>
> These sites belong to other people, and they sometimes change or go offline. Restful Booker resets its data regularly and can be slow on its first request. If a lab fails, first open the site in your browser. If the page itself has changed, that is not a problem: it is free practice for Chapter 6 (fixing broken locators). The public demo passwords used here (`secret_sauce`, `password123`) are printed on the sites themselves. They are not real secrets, and Chapter 7 still shows you how to keep them out of code.

## Code companion

All code in this book is complete and runnable. The file name appears above each code block, and the folder layout above shows where every file lives. Type it in yourself the first time; typing builds memory far better than copy-paste.

CHAPTER 1

# Why Playwright + AI Changes QA Work in 2026

*Know exactly what AI can and cannot do for a tester, and how to stay the person a team cannot do without.*

**WHAT YOU WILL LEARN**

- What has really changed in automation work, and what has not
- Why Playwright is a very good fit for AI assistants
- The key terms (LLM, agent, MCP, accessibility tree, token) in plain English
- The 5-step review loop this book uses for every AI-written test
- How to spot the typical mistakes in AI-generated test code

## 1.1 What changed, and what did not

Three years ago an automation engineer spent most of the day typing: locators, waits, page objects, boilerplate. In 2026, a good AI assistant can type a first draft of most of that in seconds. You describe a scenario, it opens a real browser, looks at the page and writes a Playwright test.

That sounds like the tester's job disappears. It does not. Look at what the AI still cannot do on its own:

- **Decide what is worth testing.** It does not know that the "Apply coupon" button caused two production incidents last quarter.
- **Know the business rule.** It sees a total of $32.39. It does not know whether tax should be 8% or 18% GST.
- **Tell a bug from a feature.** If the app is broken, the AI often writes a test that confirms the broken behaviour, and the test passes.
- **Own the result.** When a release goes wrong, "the AI wrote it" is not an answer anyone accepts.

So the work moves **up**. Less typing, more *directing* and *reviewing*. The engineers who do well in 2026 are the ones who give the AI precise instructions, read its output like a strict code reviewer, and catch the mistakes before they reach the main branch.

**A real-project example.** A retail team has 300 regression tests for a checkout flow. A redesign breaks 70 locators overnight. Without AI, two engineers spend three days fixing them. With AI and a clear review process, one engineer feeds each failure to the assistant, reviews each suggested fix, and rejects the ones that weaken an assertion. It takes a day. The AI saves typing; the engineer's judgement keeps the suite honest.

## 1.2 Why Playwright fits AI so well

Playwright is a free, open-source (Apache-2.0) framework from Microsoft for testing web apps in Chromium, Firefox and WebKit with one API. Several of its design choices make it especially good to pair with AI:

| Playwright feature | Why it helps with AI |
|---|---|
| **Auto-waiting**: actions wait until an element is visible, enabled and stable | AI-written tests need fewer hand-written waits, which are a big source of flakiness |
| **Role-based locators** such as `getByRole('button', { name: 'Login' })` | They use the same accessibility information the AI reads through MCP, so AI suggestions map directly to stable locators |
| **Web-first assertions** such as `await expect(locator).toHaveText(...)` | They retry automatically, so tests do not fail just because the page is a little slow |
| **Trace Viewer** | Records every step, DOM snapshot and network call, which you can give to an AI for failure analysis |
| **Playwright MCP** and **built-in test agents** | Official tools that let AI assistants drive a real browser and plan, write and heal tests |
| **TypeScript first** | AI models are very strong at TypeScript, and type errors catch many AI mistakes before you even run the test |

## 1.3 Key terms in plain English

You will see these words in every chapter. Learn them once here.

| Term | Plain meaning |
|---|---|
| **LLM** (large language model) | The AI "brain" that reads text and writes text, for example Claude or GPT models |
| **AI assistant** | An app that lets you talk to an LLM: a chat window, VS Code Copilot, Claude Code |
| **Coding agent** | An assistant that can also *act*: read your files, run commands, edit code, open a browser. Claude Code, OpenCode and Cline are coding agents |
| **Prompt** | The instruction you give the AI |
| **Token** | A small piece of text (about ¾ of an English word). Cloud models charge per token, and each model can only read a limited number of tokens at once (its *context window*) |
| **Hallucination** | When the AI confidently writes something that does not exist, such as a Playwright method with an invented name |
| **MCP** (Model Context Protocol) | An open standard that lets an AI assistant use external tools in a consistent way. Think of it as a "USB port" for AI tools |
| **MCP server** | A program that offers tools over MCP. **Playwright MCP** offers browser tools such as "navigate", "click" and "snapshot" |

| Term | Plain meaning |
|---|---|
| **Accessibility tree** | A structured list of what is on a page (roles like *button* or *textbox*, their names and states) that browsers build for screen readers. Playwright MCP gives this to the AI instead of a screenshot |
| **Snapshot** | A text copy of the accessibility tree at one moment |

## 1.4 What AI does well, and where it fails

Be honest with yourself and your team about both columns.

| AI is good at | AI often gets wrong |
|---|---|
| Writing a first draft of a test from a clear description | Business rules it was never told |
| Suggesting stable locators from a snapshot | Inventing APIs or config options that do not exist |
| Turning repeated code into page objects | Using `waitForTimeout()` and `force: true` to "fix" problems |
| Explaining an error message or a trace | Weakening assertions until a failing test passes |
| Generating test data and edge-case ideas | Testing that the page *loads* instead of that the feature *works* |
| Writing boilerplate: config, CI files, fixtures | Remembering your project conventions unless you write them down |

The pattern behind the right column: **the AI wants the test to pass.** You want the test to find bugs. Those two goals are not the same, and your review closes the gap.

## 1.5 The 5-step review loop

Every AI-assisted change in this book follows the same loop. Use it at work and you will rarely be embarrassed by AI code.

1. **Prompt.** Give context (app, user story, project rules) and one clear task.
2. **Generate.** Let the assistant explore the app and write code.
3. **Review.** Read every line. Check locators, waits, assertions and naming against your rules.
4. **Run.** Run the test, ideally more than once (`--repeat-each=3`). Watch it fail when you break the feature on purpose.
5. **Commit.** Commit only code you understand and would defend in a code review.

> **PRO TIP**
>
> "Watch it fail" is the step most people skip. A test that cannot fail is worse than no test, because it gives false confidence. Change an expected value on purpose, run the test, and confirm it goes red.

## 1.6 Career angle: what this means for you

Job posts for SDET and automation roles in India and the US increasingly list items like “experience with AI-assisted test generation”, “Playwright with TypeScript”, “CI/CD” and “API testing” side by side. You do not need to be an AI researcher. You need to show that you can:

- set up Playwright and an AI assistant properly,
- produce reliable tests faster *without* lowering quality,
- explain your review process and the risks clearly.

That is exactly what this book trains, and the capstone in Chapter 13 gives you a repository to prove it.

## Hands-on lab: review an AI-written test

**Goal:** find the five quality problems in a typical AI first draft, then rewrite it the Playwright way.

You can do steps 1 to 3 today, just by reading. You will run the code after the setup in Chapter 2. If Node.js is already installed, you can also run it now (step 4).

**Step 1.** Here is a test an AI assistant wrote for “add three todos” on the TodoMVC demo. Read it carefully.

**tests/ui/todo/todo-ai-draft.spec.ts**

```
// AI FIRST DRAFT - kept only for the review exercise in Chapter 1.
// It may pass, but it has five quality problems. Do not copy this style.
import { test, expect } from '@playwright/test';

test('add todos', async ({ page }) => {
  await page.goto('https://demo.playwright.dev/todomvc/');
  await page.waitForTimeout(2000);

  await page.fill('.new-todo', 'Buy milk');
  await page.keyboard.press('Enter');
  await page.fill('.new-todo', 'Write test plan');
  await page.keyboard.press('Enter');
  await page.fill('.new-todo', 'Review AI code');
  await page.keyboard.press('Enter');

  const count = await page.locator('.todo-list li').count();
  expect(count).toBe(3);
});
```

**Step 2.** Before you read on, write down every problem you can see.

**Step 3.** Compare your list with this one:

| # | Problem | Why it matters | Better |
|---|---|---|---|
| 1 | `waitForTimeout(2000)` | A fixed sleep: too slow when the app is fast, too short when it is slow. It is the number one cause of flaky tests | Remove it. Playwright auto-waits |
| 2 | CSS class selectors `.new-todo`, `.todo-list li` | Tied to styling; break when a designer renames a class | `getByPlaceholder`, `getByTestId`, `getByRole` |
| 3 | `page.fill()` and `page.keyboard` | Old page-level style; the official docs recommend locator actions | `locator.fill()` and `locator.press('Enter')` |
| 4 | `count()` + `expect(count).toBe(3)` | `count()` reads once and does not retry, so it can fail on a slow page | Web-first assertion `toHaveCount` / `toHaveText`, which retries |
| 5 | Only checks the number of items | Three *wrong* items would still pass. It also ignores the “items left” counter | Assert the exact texts and the counter |

There is also a small sixth issue: the test name `add todos` does not say what the user achieves. Good names read like a sentence: *user can add three todos*.

**Step 4.** Here is the rewrite. This file becomes part of the book’s project.

**tests/ui/todo/todo-basics.spec.ts**

```ts
import { test, expect } from '@playwright/test';

const TODO_URL = 'https://demo.playwright.dev/todomvc/';

test.describe('TodoMVC basics', () => {
  test.beforeEach(async ({ page }) => {
    await page.goto(TODO_URL);
  });

  test('user can add three todos', async ({ page }) => {
    const newTodo = page.getByPlaceholder('What needs to be done?');

    for (const item of ['Buy milk', 'Write test plan', 'Review AI code']) {
      await newTodo.fill(item);
      await newTodo.press('Enter');
    }

    await expect(page.getByTestId('todo-title')).toHaveText([
      'Buy milk',
      'Write test plan',
      'Review AI code',
    ]);
    await expect(page.getByTestId('todo-count')).toHaveText('3 items left');
  });

  test('user can complete a todo', async ({ page }) => {
    const newTodo = page.getByPlaceholder('What needs to be done?');

    await newTodo.fill('Buy milk');
    await newTodo.press('Enter');
    await newTodo.fill('Write test plan');
    await newTodo.press('Enter');

    const firstTodo = page.getByTestId('todo-item').first();
    await firstTodo.getByRole('checkbox').check();

    await expect(firstTodo).toHaveClass(/completed/);
    await expect(page.getByTestId('todo-count')).toHaveText('1 item left');
  });
});
```

`getByTestId('todo-title')` finds elements with the attribute `data-testid="todo-title"`, which the TodoMVC demo provides. `toHaveText([...])` with an array checks every item **and** their order in one retrying assertion.

**Step 5 (after Chapter 2, or now if Node.js is installed).** In a folder created with `npm init playwright@latest`, save both files under `tests/ui/todo/` and run:

```
npx playwright test tests/ui/todo --project=chromium
```

EXPECTED OUTPUT

```
Running 3 tests using 3 workers
  3 passed (4.1s)
```

Timings will differ. In Chapter 2 the project name becomes `todo`.

**Step 6. Make it fail on purpose.** Change `'3 items left'` to `'4 items left'` and run again. You should see a clear failure that shows the expected and received text. Change it back. You have now proven the test can catch a real problem.

**Step 7.** Delete the draft file. You do not want bad examples in your suite, because AI assistants copy the style of the code they find in your project.

```
rm tests/ui/todo/todo-ai-draft.spec.ts
```

## Prompt templates for this chapter

PROMPT TEMPLATE

**1. Explain a test before you trust it**

```
You are a senior Playwright reviewer. Explain this test line by line in
simple English for a tester who is new to TypeScript. Then list anything
that could make it flaky or let a real bug pass. Do not rewrite it yet.

[PASTE TEST CODE]
```

PROMPT TEMPLATE

**2. Review an AI draft against a rubric**

```
Review this Playwright test against these rules:
1. No waitForTimeout or other fixed sleeps.
2. Locators in this order: getByRole, getByLabel, getByPlaceholder,
   getByText, getByTestId. CSS/XPath only with a written reason.
3. Only web-first assertions (await expect(locator)...).
4. Assertions must prove the feature works, not just that the page loaded.
5. Test names describe user behaviour.
For each problem give: line, rule broken, why it matters, the fixed line.
Only use Playwright APIs that exist in @playwright/test [VERSION].

[PASTE TEST CODE]
```

## Common mistakes and how to fix them

- **Accepting the first draft.** Fix: always run the 5-step loop. Treat AI code exactly like a pull request from a new joiner.
- **Judging a test by whether it passes.** Fix: break the feature (or the expected value) and confirm the test fails.
- **Leaving bad examples in the repo.** Fix: delete or rewrite them. Agents learn your style from your existing files.
- **Believing "AI will replace testers" or "AI is useless."** Both are wrong. Fix: measure. Track time saved and bugs found, and decide with data.

## Pro tips from real projects

> **PRO TIP**
>
> Keep a short **"AI mistakes we have seen"** list in your team wiki: invented APIs, sleeps, weakened assertions. Review it in every retro. New joiners learn the review habit in a week.

- Use AI first on the boring, well-defined work (locator fixes, page objects, test data), then move to test design once your review habit is strong.
- When an AI suggests an API you have never seen, search playwright.dev for it before you accept it. This takes ten seconds and catches most hallucinations.

## Interview questions

**Q1. Will AI replace automation testers?**

No. It replaces a lot of *typing*, not the judgement. AI does not know business rules, cannot tell a bug from a feature, and tends to write tests that pass instead of tests that find problems. My role shifts towards deciding what to test, writing precise instructions, and reviewing AI output strictly. That makes a good tester more productive, not unnecessary.

**Q2. Why is Playwright a good fit for AI-assisted testing?**

Auto-waiting and web-first assertions remove most manual waits, which are where AI code usually goes wrong. Role-based locators use the same accessibility tree that Playwright MCP gives the AI, so the AI's view of the page maps straight to stable locators. Trace Viewer gives rich failure data to analyse, and Playwright ships official MCP and test-agent tooling.

**Q3. What is MCP, in one minute?**

Model Context Protocol is an open standard for connecting AI assistants to external tools. An MCP server exposes tools; the assistant calls them. Playwright MCP exposes browser tools such as navigate, click, type and snapshot, so the assistant can drive a real browser and read the page as a structured accessibility tree instead of guessing from a screenshot.

**Q4. What are the most common problems in AI-generated tests?**

Fixed sleeps, brittle CSS selectors, invented APIs, non-retrying checks like `count()` followed by a plain `expect`, and weak assertions that only prove the page loaded. The most dangerous one is an AI “fix” that weakens an assertion so a failing test passes and hides a real bug.

**Q5. How do you make sure an AI-written test is trustworthy?**

I follow a loop: prompt with context, generate, review every line against our rules, run it several times, and break the feature on purpose to see it fail. Only then do I commit, and the pull request still goes through normal code review.

## Chapter checklist

- [ ] I can explain what changed in QA work and what did not
- [ ] I can define LLM, coding agent, MCP, accessibility tree and token
- [ ] I can list five common problems in AI-generated test code
- [ ] I know the 5-step loop: Prompt, Generate, Review, Run, Commit
- [ ] I rewrote the AI draft and understand every line of `todo-basics.spec.ts`
- [ ] I made a test fail on purpose and saw a clear error