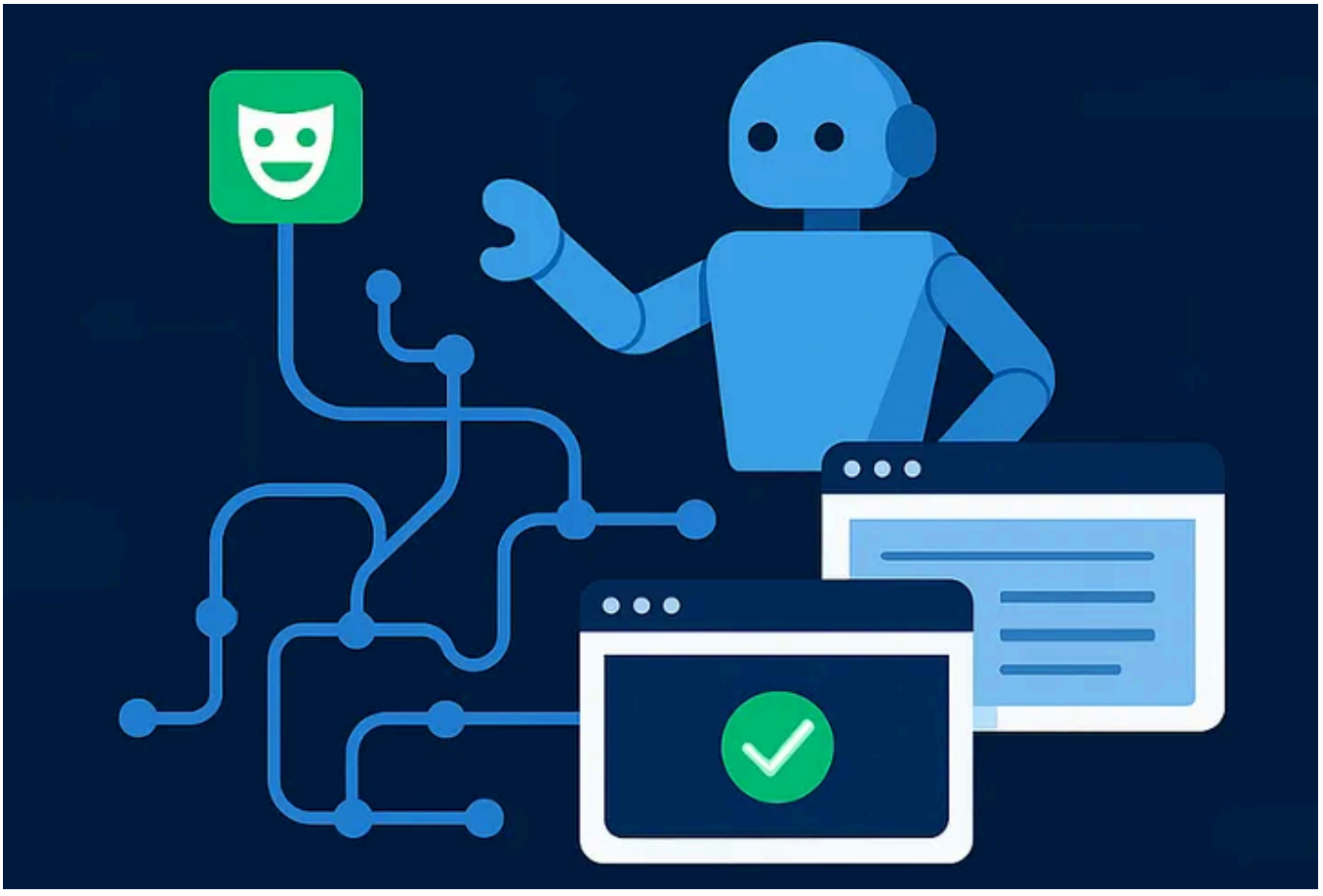
How Playwright Test Agents Are Changing the Game in E2E Automation

16 min read · Oct 27, 2025

Kostiantyn Teltov Follow



Hi colleagues,

A few weeks ago, the Playwright team took another step forward in the evolution of MCP with the release of three new Playwright agents. Only a lazy didn't mentioned about it. I tried them out shortly after launch but decided to wait a bit until the initial

hype settled down. Of course, I couldn't miss the opportunity entirely — and now I'd like to share some thoughts about it.

Why it is so hyped? Playwright already had a release of MCP server before. So, what is the difference? Let's start from the beginning and try to understand what is MCP server and what are these 3 agents about.

. . .

🧠 MCP server and Playwright agents

First, let's try to understand what is MCP server in general.

MCP server without water

Model Context Protocol (MCP) is an open standard for letting AI apps/agents call external capabilities in a **consistent and secure** way.

In simple words: you have an LLM(Large Language Model) and you want it to **use tools** (do things) and **read resources** (get context). **MCP** is the bridge that makes that integration predictable, permissioned, and inspectable.

Core pieces

- **MCP client** — the AI application/agent (desktop app, IDE, chat UI, CI bot).
- **MCP server** — the component that exposes **tools** and **resources** over MCP.
- **Tools** — parameterized actions the agent can call (e.g., `edit/createFile`, `playwright-test/browser_navigate`).
- **Resources** — read-only (or narrowly scoped) data the agent can fetch (e.g., `files://specs/*.md`).
- **Sessions** — scoped interactions with lifetimes, policies, and logs (think: a sandbox + audit trail).
- **Transport** — typically JSON-RPC over stdio; some servers also support HTTP/SSE or WebSockets.
- **Security** — explicit tool/resource registration, input validation, rate limits, redaction/logging, and opt-in access.

Playwright MCP

Playwright MCP is a **Model Context Protocol (MCP) server** that brings Playwright's browser automation to AI agents. It lets large language models (LLMs) interact with web pages in a structured way — using accessibility data instead of screenshots. This

means the AI can understand and control websites more reliably, without depending on visual or pixel-based models.

Playwright Agents

While **Playwright MCP** provides the protocol and browser automation foundation, **Playwright Test Agents** take it a step further — turning automation into *intelligent collaboration* between AI and Playwright.

These agents are preconfigured AI-driven components built on top of the Playwright MCP server. Each agent can understand goals, plan browser actions, and execute them autonomously through structured API calls — instead of running predefined scripts.

In simple terms, **Playwright MCP is the engine**, and **Playwright Test Agents are the drivers**.

Why It Matters?

Previously, you had to write every step manually — "go to page," "click button," "assert text." Now, with MCP and Playwright agents, you can ask the system to *figure out* and *execute* those steps for you.

It's not just automation — it's **AI-assisted testing**, where human engineers set the goals and the agent handles the tedious browser work.

Details

Starting from **Playwright v1.56**, three new **agents** were introduced:

🎭 **Planner** — explores the application and produces a **Markdown test plan.**
🎭 **Generator** — converts the Markdown plan into **Playwright Test files.**
🎭 **Healer** — executes the test suite and **automatically repairs failing tests.**

If we look at it from a **classical test automation workflow**, the first step is to **design test scenarios or test cases** to be automated.

🎭 **Planner**
That's exactly the responsibility of the **Planner Agent** — it explores the application and generates a detailed **test plan in Markdown (.md) format.**

```
# TodoMVC Application - Basic Operations Test Plan
## Test Scenarios
### 1. Adding New Todos
**Steps:**
1. Click in the "What needs to be done?" input field
2. Type "Buy groceries."
```

```
3. Press Enter key
**Expected Results:**
- Todo appears in the list
- Counter shows "1 item left."
```

🎭 Generator

Next, we create the **automated test scenarios.**
The **Planner Agent** takes the previously generated plan and turns it into **Playwright E2E scenarios.** It can produce both **UI** and **API** tests.

Example scope:

- **UI:** navigate, interact with elements, assert text/URLs/states.

- **API:** send requests, validate status codes, schemes, and responses.

```
import { test, expect } from '@playwright/test';
test('Add valid todo', async ({ page }) => {
   await page.fill('#new-todo', 'Buy groceries');
   await page.press('#new-todo', 'Enter');
   const todoText = await page.textContent('.todo-list li');
   expect(todoText).toBe('Buy groceries');
});
```

🎭 Healer

Of course, even the best test design and implementation aren't always perfect — and the same applies to Playwright agents.
That's why the **Healer Agent** was created: it automatically **fixes failing tests** and **runs**

them again. This helps keep your test suite robust, even when UI elements change, and minimizes downtime caused by broken tests.

You may use these agents as a part of source code repo or as stand-alone e2e test repository. But you should remember LLM works better if it has more context. Logically with access to the code it might have much more of context. It means, we might change a classical way of creation E2E test automation solutions from stand-alone repo to be a part of source code repo.

Another cool thing about these agents is that they work **independently**.
You can use them **together as a chain** — Planner → Generator → Healer — or **run only the one you need** for a specific task.

But enough theory — let's set them up and see them in action.

. . .

⚙️ Playwright agents setup

Pre-requisites

To use **Playwright Agents**, you'll need the following:

```
Node.js (v18 or later)
IDE: Visual Studio Code, Claude Code, or Opencode
Copilot: preferably with Claude Sonnet (latest, e.g. 4.5 at the time of writing
Playwright v1.56 or later (actually, we will install it after)
```

Installation

1. We need to install Playwright. In my case, I work with VS Code. So, let's continue steps with this IDE.

```
npm init playwright@latest
```

If you look to Copilot chat it does not contain any specific agents

2. Install Playwright agents

```
npx playwright init-agents --loop=vscode
```

Copilot chat was extended with 3 agents

In addition to the chat mode files, several **agent files** were created. These files play a key role — they tell the IDE (like Claude Code, VS Code Copilot, or Opencode) how to communicate with each Playwright agent.

Each file (e.g., `planner.chatmode.md`, `generator.chatmode.md`, `healer.chatmode.md`) defines **how your AI copilot interacts with a specific Playwright MCP agent** inside your IDE.

And the last step — make sure the `mcp.json` file is present in your project.

This file defines how your IDE connects to the **Playwright MCP server**. It specifies the command used to start the MCP process and allows your Copilot or IDE (like **Claude Code** or **VS Code**) to communicate with it.

Configuration

Our goal now is to configure the necessary tools for our agents.

Make sure that you don’t have the “Update tools” option available, as shown in the screenshot below. If so, click it to update.

Then, enable the tools. You may enable all of them at this time.

If you open 🎭 `planner.chatmode.md`, you'll notice that your tools have been updated with the selected tool items.

💡 **Note:** When I first tried this a few weeks ago, it didn't work on my machine — I had to manually add the tool names to the file. Fortunately, it seems this issue has since been fixed.

Let's configure the tools for the other agents so that we don't waste time in the future. The process will be the same. You just need to open the agent configuration and enable the tools.

Using Seed files

When you work with **Playwright Agents**, you might notice a file called `seed.spec.ts` inside your project.
This file isn't part of the standard Playwright setup — it's created to help the **agents** (Planner, Generator, and Healer) get started.

The `seed.spec.ts` file is used to **seed new tests** and is **copied into generated tests** by the agent. It can be left empty, but it's often used to include **fixtures or setup logic** that need to run **before the actual tests**.

As an example, we would like to start with some already logged page as precondition.

```
test.describe('seed for logged in user', () => {
  test('seed using listPage fixture', async ({ listPage }) => {
    const page = listPage; // set the page to the listPage fixture
  });
});
```

. . .

🗺️Playwright agents usage — Planner

I would like to create a test plan for the search function of the Retro Games Portal.

Let's start with the first prompt. I would also like to provide the repository's source code.

```
Generate test plan for search functionality of the website http://localhost:900
Source code of the website https://github.com/dneprokos/retro-video-games-porta
```

When you see the message **"Setup planner page" and playwright-test(MCP Server)** during startup — congratulations, your setup worked!
If you don't see it, go back and verify your configuration steps; something might still be missing.

Next, the agent will start **exploring your application** and **creating a test plan**.
It may ask you to continue or perform certain actions during the process.

💡 **Note:** Be careful — if your local environment doesn't have the necessary permissions, the agent might not be able to create the plan successfully.

Next phase is to review this test plan. Test design looks great. I would like to show you only some parts.

```
# Retro Video Games Portal - Search Functionality Test Plan

## Executive Summary

This test plan covers the search functionality of the Retro Video Games Portal

---

## Test Scenarios

### 1. Search for a Game by Full Title

**Assumptions:** Fresh homepage, no filters applied.

**Steps:**

1. Focus on the "Search games..." textbox.
2. Type the full name of a game (e.g., "Contra").
3. Observe the game list.

**Expected Results:**

- Only games matching "Contra" are displayed.
- The game "Contra" appears in the results.
- Pagination updates to reflect filtered results.

---

### 2. Search for a Game by Partial Title

**Assumptions:** Fresh homepage.

**Steps:**
```

```
1. Focus on the "Search games..." textbox.
2. Type a partial name (e.g., "Man").
3. Observe the game list.

**Expected Results:**

- All games with "Man" in their title (e.g., "Mega Man 2", "Castlevania") are d
- Non-matching games are hidden.

---
.....................................
.....................................
```

Here you need to decide how to act next. On my opion some of the test plan scenarios can be moved on Unit/Integration tests layer. That means we don't want to implement E2E tests for these scenario.

Logically, my next step was to ask about updating our tests to be split on a multiple layers.

```
Can you mark some of the scenarios with Unit/Integration test layer
in order to avoid implementing them on e2e test layer.
```

That's where the problems started to appear.
Sometimes, the agent marked certain tests as belonging to the **Unit Test** layer, even when it didn't make sense for those cases. I had to review the plan multiple times to make sure each test was assigned to the correct layer.

> AI can make us lazy — but remember to stay disciplined and always verify the output.

Let's see some of the final plan example

```
## Test Scenarios

### 1. Search for a Game by Full Title
```

```
**Assumptions:** Fresh homepage, no filters applied.

**Test Layer:** E2E

**Steps:**

1. Focus on the "Search games..." textbox.
2. Type the full name of a game (e.g., "Contra").
3. Observe the game list.

**Expected Results:**

- Only games matching "Contra" are displayed.
- The game "Contra" appears in the results.
- Pagination updates to reflect filtered results.

---

### 2. Search for a Game by Partial Title

**Assumptions:** Fresh homepage.

**Test Layer:** Integration (API)
**API Endpoint:** `GET /api/games?search=Man`

**Steps:**

1. Focus on the "Search games..." textbox.
2. Type a partial name (e.g., "Man").
3. Observe the game list.

**Expected Results:**

- All games with "Man" in their title (e.g., "Mega Man 2", "Castlevania") are d
- Non-matching games are hidden.

---
.........................
.........................
```

And we are ready to move to test generator

. . .

Playwright agents usage — Generator

Next I will switch an agent to “generator”

Also it is recommended to work in a clear context for the new agent. I will clean up chat too.

Now we're ready to cook. I'll provide a prompt to generate both **E2E** and **API** tests *(and in this context, API tests are effectively E2E as well)*.

```
Generate tests E2E and Integration (API) tests based on search-test-plan.md tes
Source code of the website https://github.com/dneprokos/retro-video-games-porta
Local website: http://localhost:9000
API: http://localhost:9000/api/
```

Generator started to run and that means we are on the right track.

After some time tests generation was finished

A multiple files with automated tests were created. At first glance it looks not bad. It split it different functionality

```
// spec: search-test-plan.md

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

test.describe('Search Functionality', () => {
  test('Search for a Game by Full Title', async ({ page }) => {
    // 1. Navigate to homepage
    await page.goto('http://localhost:9000/');

    // 2. Focus on the search textbox and type "Contra"
    const searchBox = page.getByPlaceholder('Search games...');
    await searchBox.click();
    await searchBox.fill('Contra');

    // 3. Verify filtered results
    await expect(page.getByText('Contra')).toBeVisible();

    // 4. Verify other games are not visible
    await expect(page.getByText('Super Mario')).not.toBeVisible();

    // 5. Verify pagination update
    const paginationInfo = page.locator('.pagination-info');
    await expect(paginationInfo).toBeVisible();
  });
});

// Integration tests
test.describe('Search API Integration', () => {
  test('Search for a Game by Full Title - API', async ({ request }) => {
    const response = await request.get('http://localhost:9000/api/games?search=
    expect(response.ok()).toBeTruthy();

    const data = await response.json();
    expect(data.length).toBeGreaterThan(0);
    expect(data[0].title).toContain('Contra');
  });

  test('Search with No Results - API', async ({ request }) => {
    const response = await request.get('http://localhost:9000/api/games?search=
    expect(response.ok()).toBeTruthy();

    const data = await response.json();
    expect(data.length).toBe(0);
  });
});
```

It is time to run these tests. For simplicity I will update my “playwright.config.ts” to run tests only with “Chromium” browser.

And let’s run our tests

```
npx playwright test
```

Honestly didn’t expect all the tests will fail.

But it might be even interesting. Because were are planning to move to our last agent called “Healer”.

. . .

💉 Playwright agents usage — Healer

We will use the same approach as before. We will select the ‘Healer’ agent and clean up our chat session.

Prompt

```
Heal all the tests.
Source code of the website https://github.com/dneprokos/retro-video-games-porta
```

```
Local website: http://localhost:9000
API: http://localhost:9000/api/
```

Ok. UI tests were fixed quickly. As for API tests it was many rounds to fix. I didn't measure, but I believe it took more than 30 minutes to do all the fixes.

Please remember all of these iterations will use paid tokens if you have a paid version of copilot.

Here is an example of fixed tests file.

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

test.describe("Search Functionality", () => {
  test("Search for a Game by Full Title", async ({ page }) => {
    // 1. Navigate to homepage
    await page.goto("http://localhost:9000/");

    // 2. Focus on the search textbox and type "Contra"
    const searchBox = page.getByPlaceholder("Search games...");
    await searchBox.click();
    await searchBox.fill("Contra");

    // 3. Verify filtered results
    await expect(page.getByText("Contra")).toBeVisible();

    // 4. Verify other games are not visible
    await expect(page.getByText("Super Mario")).not.toBeVisible();

    // 5. Verify pagination update
    const paginationInfo = page.getByText(/Showing \d+ of \d+ games/);
    await expect(paginationInfo).toBeVisible();
  });
```

```
});

// Integration tests
test.describe("Search API Integration", () => {
  test("Search for a Game by Full Title - API", async ({ request }) => {
    const response = await request.get(
      "http://localhost:9000/api/games?search=Contra"
    );
    expect(response.ok()).toBeTruthy();

    const responseData = await response.json();
    expect(responseData).toBeTruthy();
    expect(responseData.games).toBeTruthy();
    expect(responseData.games[0]).toEqual(
      expect.objectContaining({
        name: expect.stringContaining("Contra"),
      })
    );
  });

  test("Search with No Results - API", async ({ request }) => {
    const response = await request.get(
      "http://localhost:9000/api/games?search=nonexistent"
    );
    expect(response.ok()).toBeTruthy();

    const responseData = await response.json();
    expect(responseData).toBeTruthy();
    expect(responseData.games).toBeTruthy();
    expect(responseData.games.length).toBe(0);
  });
});
```

The **quality of the generated code** is, of course, **questionable** — and some of the **assertion steps** may raise concerns. Still, we can always review and refine them, or prompt the agent to add additional assertions whenever deeper verification is needed.

Finally, I asked the agent to generate the `README.md` and `.gitignore` files so I could push the project to **GitHub**.

💎 Bonus — Page Object and API helpers

I'll be honest I was not happy with this plain spahetti code generated by the agents. So, I decided go beyond and try to ask about minimum things to improve. I switched to generator agents and my prompt was pretty straight and simple.

```
Create pages and api and implement. Re-use in the tests
```

Job was done. Here is an example

PageObject:

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

export class SearchPage {
    readonly page: Page;
    readonly searchBox: Locator;
    readonly paginationInfo: Locator;

    constructor(page: Page) {
        this.page = page;
        this.searchBox = page.getByPlaceholder('Search games...');
        this.paginationInfo = page.getByText(/Showing \d+ of \d+ games/);
    }

    async navigateToHome() {
        await this.page.goto('/');
    }

    async searchForGame(gameName: string) {
        await this.searchBox.click();
        await this.searchBox.fill(gameName);
    }

    async verifyGameVisible(gameName: string) {
        await expect(this.page.getByText(gameName)).toBeVisible();
    }


..............................
}
```

API client:

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

export class GamesApiClient {
    readonly request: APIRequestContext;
```

```
    constructor(request: APIRequestContext) {
        this.request = request;
    }

    async searchGames(searchTerm: string) {
        const response = await this.request.get(`/api/games?search=${searchTerm
        expect(response.ok()).toBeTruthy();
        return await response.json();
    }

    async verifyGameInResults(responseData: any, expectedGameName: string) {
        expect(responseData).toBeTruthy();
        expect(responseData.games).toBeTruthy();
        expect(responseData.games[0]).toEqual(
            expect.objectContaining({
                name: expect.stringContaining(expectedGameName),
            })
        );
    }

    async verifyNoResults(responseData: any) {
        expect(responseData.games).toHaveLength(0);
    }
}
```

Refactored test file:

```
import { test } from '@playwright/test';
import { SearchPage } from './pages/SearchPage';
import { GamesApiClient } from './api/GamesApiClient';

test.describe('Search Functionality', () => {
    let searchPage: SearchPage;
    let apiClient: GamesApiClient;

    test.beforeEach(async ({ page, request }) => {
        searchPage = new SearchPage(page);
        apiClient = new GamesApiClient(request);
    });

    test('Search for a Game by Full Title', async () => {
        // 1. Navigate to homepage
        await searchPage.navigateToHome();

        // 2. Search for game
        await searchPage.searchForGame('Contra');
```

```
        // 3. Verify filtered results
        await searchPage.verifyGameVisible('Contra');

        // 4. Verify other games are not visible
        await searchPage.verifyGameNotVisible('Super Mario');

        // 5. Verify pagination update
        await searchPage.verifyPaginationVisible();
    });
});

test.describe('Search API Integration', () => {
    let apiClient: GamesApiClient;

    test.beforeEach(async ({ request }) => {
        apiClient = new GamesApiClient(request);
    });

    test('Search for a Game by Full Title - API', async () => {
        const responseData = await apiClient.searchGames('Contra');
        await apiClient.verifyGameInResults(responseData, 'Contra');
    });

    test('Search with No Results - API', async () => {
        const responseData = await apiClient.searchGames('nonexistent');
        await apiClient.verifyNoResults(responseData);
    });
});
```

It’s not perfect, but it looks much better now. And of course, you can keep refactoring it — either with the help of agents or manually.
The most important thing is to **understand what you’re doing and why**.

And now it is time to make some conclusions

. . .

🌍 Final Thoughts

Let's be honest — these three **Playwright Agents** look really impressive.
They represent another big step forward in the evolution of **E2E test automation**.

Here are some of my personal thoughts on their **advantages** and **disadvantages** 👇

Advantages:

- **High-quality test cases** — even though we didn't test very complex scenarios, the generated structure looks solid and well-organized.
- **Less manual coding** — a lot of repetitive work is handled automatically.
- **Fast creation** — the speed of generating and running tests is impressive.
- **Auto-healing** — the Healer Agent can fix failing tests automatically, saving valuable time.
- **Flexible usage** — each agent can work **independently** or as part of a **combined workflow**.

Disadvantages:

- **Code quality** — the generated code may not always follow best practices or be production-ready.

- **Review time** — human validation is still required; the agent can produce results that need attention.
- **Complacency risk** — it's easy to start trusting the AI too much and skip double-checking.
- **Human understanding still matters** — you still need to know **what to build**, **how it should work**, and which **design patterns** to apply if you want clean, maintainable code.
- **Mindset shift for experienced engineers** — seasoned testers and developers need to adjust. You no longer write as much code yourself — instead, you **guide**, **review**, and **shape** what the agents create.

Project example

We truly are in a **brave new world**.
LLMs are already part of our lives, and they're here to stay — growing smarter and more capable with time.

So we need to **adapt**. And adapting doesn't mean resisting change — it means **learning how to use it wisely**.
But let's be honest, I'm not saying anything new. We all know the rule: **use AI smartly**. Don't let it make you lazy.
Discipline yourself to **verify the results**, question what it produces, and stay in control.

These tools can **dramatically speed up our work**, but we still need to **understand what we're doing** and make sure we're **aligned with the generated output.**

Set sail — and good luck on your journey.
Hopefully, we'll all find our **promised land** somewhere along the way. ⚓

Playwright Test | Testing | Test Automation | AI | QA

Follow

Written by Kostiantyn Teltov

1.1K followers · 104 following

From Ukraine with love. QA Tech Lead / SDET / QA Architect (C#, JS/TS, Python, Java). Passionate about testing, teaching, and dreaming of making indie games.

Responses (5)

Bgerby

What are your thoughts?

Akash
6 days ago

it was very helpful

1 reply Reply

Sharmapriyans
6 days ago

A very insightful article! Playwright Test Agents are truly revolutionizing E2E automation by offering faster execution, better scalability, and smarter testing workflows. It’s exciting to see how this innovation is reshaping the future of automated... more

Reply

Keerthivasan he/him
20 hours ago

That's awesome Teltov ! I have been using MCP for a while now , may be I'll have to try out genrator and healer. Been using generic prompt.md file to create tests , will integrate generator and healer into my framework. Thanks !

Reply

See all responses

More from Kostiantyn Teltov

Kostiantyn Teltov

Layered Test Automation with Cursor AI: Following the Testing Pyramid

Hi Dear Reader,

Sep 29 2

Kostiantyn Teltov

SDET: Introduction to the first Playwright project

I’m sorry, some people have already told me that I like AI-generated images. I would say that I like how they make your life easier. I used...

Mar 20, 2024 4

Kostiantyn Teltov

Playwright with .NET. What? Is it useful?

Greetings, my fellow coding nerds, Today I decided to try to research and write articles in parallel. So this is a kind of lab experiment...

Sep 24, 2023 2

Kostiantyn Teltov

Design Patterns for QA Automation: Build effective test solutions

phew! Good. Finally I have finished this article.

Jul 25, 2022 10

See all from Kostiantyn Teltov

Recommended from Medium

Shahnawaz Khan

🚀 AI Agents Are Writing Playwright Tests Now—Here's What That Means for QA Engineers

Once upon a time, test automation meant writing selectors, assertions, and flows by hand then came frameworks like Cypress & Playwright ... ...

✨ Nov 1

Manish Saini

Shift-Left is Dead. Welcome to “Shift-Everywhere” Testing.

For years, “Shift-Left Testing” was the cornerstone of modern QA strategy. The mantra was simple—test early, test often, and stop defects...

Oct 24

KailashPathak

Smart Way to Generate Locators for Playwright and Selenium

In modern test automation, finding the right locator is half the battle. Whether you’re using Playwright or Selenium, reliable locators are...

Oct 30 2

In Towards AI by Teja Kusireddy

We Spent $47,000 Running AI Agents in Production. Here's What Nobody Tells You About A2A and MCP.

Multi-agent systems are the future. Agent-to-Agent (A2A) communication and Anthropic's Model Context Protocol (MCP) are revolutionary. But...

Oct 16 111

George Ukkuru

Why Only 10% of Testers Will Remain by 2030

Every few years, people in tech love to ask a scary question: Is my job going to disappear? For those of us in software testing, this...

Oct 26 1

In JavaScript in Plain English by henry messiah tmt

Top 10 Test Automation Frameworks for JavaScript Developers (2025 Edition)

Introduction

Oct 30

See more recommendations