

Chapter 4. Automated Testing and Quality Assurance

Testing and quality assurance (QA) are usually the last gates that new software code must pass through before it gets deployed in production. Their ultimate goal is to find costly bugs or other standout issues that may have made it through code review (as covered in the previous chapter) to avoid putting them into production.

The QA process happens after code has been developed, reviewed, and accepted to merge into the codebase. There is occasional confusion between testing and QA as concepts, perhaps because the stakeholders traditionally involved are called either testing engineers or QA engineers at different companies. Whatever the title, though, they are usually in charge of the process covered in this chapter.

Typically, the QA process consists of conducting manual and/or automated tests in an environment that closely matches production and mimics user behavior, to catch any bugs that escaped the code-review process.

When such bugs are found during testing/QA, the feature is regressed back to development status. The original software engineer in charge of implementation must fix the issues before pushing the feature to review and QA again. These regression loops aim to guarantee that the code that ultimately gets deployed to production is indeed bug-free.

These processes are critical to any software development team. We can break them into two main categories: automated and manual.

Automated tests

Automated testing employs specialized software tools to execute pre-specified tests on the application. This method is highly efficient for repetitive and regression tests, because it reduces the time needed to validate new code changes. Automation ensures consistency and precision, minimizes the risk of human error, and enables extensive

test coverage. Automated tests can run around the clock, providing rapid feedback and allowing for continuous integration and continuous delivery (CI/CD) pipelines. Although initially setting up automated tests requires effort, as does maintaining them, the long-term benefits include faster release cycles, improved accuracy, and the ability to quickly detect and address defects.

Manual tests

In manual testing, human testers meticulously execute test cases without the assistance of automated tools. They simulate end-user behavior to identify defects, ensuring that the software behaves as expected in real-world scenarios. This approach allows for nuanced understanding and adaptability, often catching issues that automated scripts might miss, such as user-interface glitches and usability concerns. While manual testing can be time-consuming and labor-intensive, it remains essential for exploratory testing, where creativity and intuition are crucial in uncovering unexpected bugs and ensuring a seamless user experience.

QA is a meticulous, careful process by its nature, which often makes it a bottleneck that delays features going live. As such, there's a market for AI tools that propose to accelerate different parts of this process. This chapter will focus on two of those tools in particular.

AI is changing every aspect of automated testing. For example, until very recently, automating testing involved writing complex scripts. Now, however, many automated testing tools provide ways to create tests without writing a single line of code. With simple, plain English, you can create automated tests that check every component and functionality in your software application. Visual testing has also been simplified with AI-powered tools that automatically detect visual bugs, ensuring that your user interface looks and works as intended. These improvements make the testing process more effective and efficient, which allows testers to focus on improving the overall quality of the software.

Types of AI Testing Tools

In addition to the automated/manual divide, we can also classify AI tools for software testing and quality assurance as *functional* and *nonfunctional*, based

on the specific areas they target within the testing lifecycle.

Functional AI testing tools

As the name implies, *functional* testing tools verify that a software application performs all of its intended functions accurately. These tools focus on what the system does. Their goal is to test whether the application's internal components deliver the expected output.

Functional testing tools handle unit tests, integration tests, visual tests, regression tests, and smoke tests, for example.

Nonfunctional AI testing tools

Nonfunctional AI testing tools assess aspects of software that go beyond its functional behavior, such as its performance, compatibility, usability, security, and reliability. These tools focus on evaluating the software's *performance* rather than its behavior. They measure speed, response time, and resource utilization, to name a few.

Tools in both categories aim to identify potential performance issues and security vulnerabilities. They use deep learning models trained on customer usage data, internal company documents, or even industry regulatory norms or standards. These algorithms can learn to identify patterns that may indicate performance bottlenecks or security risks. This underlying “intelligence” makes these AI tools important peers of humans in the QA stage of the software development lifecycle. The biggest gain to be reaped from using these tools is that they can apply their intrinsic testing acumen on large codebases in near-real time.

A common frustration is that QA takes a long time, since complex products and extensive codebases usually have hundreds of different user journeys to test, and doing this manually is very time-consuming. Automated tools do not reduce the value of having a human in the loop, but they can certainly automate a lot of repetitive work, freeing human QA professionals to focus on the critical flows, ones that were changed in the last pull request, or whatever makes up the 20% of work that has 80% of the impact (as per the [Pareto principle](#), so often used in software development).

Many of the prominent tools I evaluate in this chapter combine functional and nonfunctional testing abilities, as they aim to integrate into various development environments. These tools can be used in different ways,

depending on each team's context and preferences. For instance, testing is one of the most significant aspects of the CI/CD process. Thanks to CI/CD-integrated testing tools, we now conduct tests continuously during development rather than waiting until after development. This continuous integration approach provides real-time feedback about your software's performance and internal functioning.

CI/CD-integrated AI testing tools automatically test changes made to your code after every build. Continuous testing ensures that issues are identified and addressed early in the development cycle, reducing the risk of defects in production. This approach promotes a culture of quality and allows for faster, more reliable software releases.

In contrast, browser and cloud-based tools run tests in web browsers or the cloud, providing flexibility and accessibility. They allow testing on different devices and environments, without complex setups like IDEs and CI/CD-integrated tools.

Use Cases

Software developers and engineering teams across various industries are integrating AI testing and QA tools into their processes. Here are some of the prominent use cases that we've seen:

Automated test creation

Building test automations used to be very slow and time-consuming. It takes a lot of time and mental bandwidth to design and write test scripts, run regression tests, and do everything in between. This is what many AI-driven testing tools aim to help with, by generating comprehensive test scripts from plain English prompts within seconds. This natural language processing (NLP) method of scripting makes it easy to automate complex workflows. This, in turn, makes testing accessible to both technical and nontechnical stakeholders. AI-generated test scripts are usually based on user behavior and existing patterns in previous test data, which makes the tests more relevant and closer to what a human QA tester would create.

Improved test accuracy

Improving accuracy means fewer code bugs slip through the QA stage to production. AI algorithms' superpower is that, unlike manual testers, they can capture patterns and anomalies at scale. Being trained on extensive codebases and past testing data helps them more easily spot nuances that might indicate an issue requiring the feature to be regressed.

Self-healing capabilities after encountering errors

AI testing tools with *self-healing* capabilities automatically detect and fix issues in test scripts when changes in the application's UI or code cause tests to fail. This ensures that all tests remain functional and up to date without manual intervention. Historically, updates are one of the biggest challenges for QA teams, since a change in the UI means many tests written in the past also need to be changed. These AI tools can significantly reduce the maintenance burden on QA teams and make the QA process faster and more reliable.

Faster software-release cycles

By automating repetitive tasks using AI testing tools, we can speed the release cycle of software applications tenfold. Developers can concentrate more of their time on innovating new features and enhancing the product instead of spending the entire day trying to catch bugs or write test scripts. Companies can also respond faster to market demands and user feedback.

The Need for Human Testers

It is important to remember that while these AI tools can do a great job catching issues and bugs that would eventually break production, the human instinct is still crucial during testing. This is not just about the limitations of the tools reviewed here, nor their underlying AI algorithms. It goes beyond that. Software development teams don't write 100% of their requirements and edge cases in an absolutely perfect way.

I can speak from my own experience leading software teams for more than a decade: there are *always* changes and caveats based on last-minute user feedback, an ad hoc request from sales, or even a phone call from the CEO with a specific exception. While teams try hard to properly document all

requirements and capture edge cases and test plans in the software development task descriptions, the result is never perfect. There are gaps. And because these are the written materials on which AI tools are trained, and they take project requirements as the ultimate instructions to test against, they'll eventually miss some nuances of those requirements or ad hoc exceptions.

Even beyond that, frankly, there's often specific context awareness that only humans can have. We need humans in order to adapt to industry-wide events or sensitive user concerns. Software development is a complex matter, and the more extensive a product and codebase are, the more likely it is that a purely AI-driven QA process will show its limitations and gaps.

AI algorithms are only as good as the data used in training them. They can absolutely help a lot, as this chapter shows—especially with the repetitive grunt work, like testing an extensive list of user journeys and application flows. But human monitoring, review, and intervention are still needed for the critical parts of the process.

Evaluation Process

Most companies in the QA automation space cater to enterprise clients. This makes sense, given that enterprise companies tend to have larger teams, more extensive products, and much higher quality-control standards. While this is totally fine and expected, it affected my selection process for tools to showcase here, since I gave preference to tools that can be accessed via a simple self-service sign-up process and that offer a free trial. This is a deliberate choice to make it easier for readers to act on what they read here, though it certainly leaves out some tools that required me to speak with their sales teams to negotiate a price package. I decided those tools were out of scope for this book.

Even with that limitation, as I researched this chapter, I reviewed more than 20 automated testing tools (many of which fell into that enterprise sales category). I shortlisted the two tools highlighted next.

To evaluate and compare AI-powered testing tools for this chapter, I used each tool to write and run test cases for [a simple, straightforward test site](#): a basic web application for booking appointments with a medical doctor. Since developing a comprehensive, end-to-end automated testing framework is a

substantial undertaking, I focused on evaluating the specific AI features these testing tools offer, to demonstrate their potential for integrating AI into software testing.

The examples in this book are not intended to represent a complete testing framework, but rather to demonstrate how to use AI-integrated features in automated testing tools. The primary objective of this chapter is to showcase AI's possibilities and simplicity in the software testing domain, not to provide a production-ready solution.

I evaluated how the AI features in these tools enhance various aspects of the testing process, such as generating test cases, creating test data, executing tests, and analyzing results.

Test site

<https://katalon-demo-cura.herokuapp.com>

App description

Web app with a login page for booking appointments with a medical doctor

Test description

Automate a series of actions on a healthcare service website. This test ensures that a patient can navigate the app and use it to successfully book an appointment to meet with the doctor. We want to see if everything works as it should on the app.

Steps

We intend to generate/create test cases that automatically evaluate whether:

- The login page works perfectly
- Users can successfully book appointments if all the required fields are updated
- The booking history records every booking made

Test case 1

1. Navigate to <https://katalon-demo-cura.herokuapp.com>.
2. Click on the Make Appointment button.
3. Set the text John Doe in the username field.
4. Set encrypted text in the password field.
5. Click on the Login button.
6. Check if the user can successfully log in when the correct information is entered.

Test case 2

On the Make Appointment page:

1. Select a Visit Date.
2. Select the Medicare option.
3. Select the “Apply for hospital readmission” option.
4. Enter a comment in the text area.
5. Book an appointment.
6. Check that the user can successfully book an appointment 10 seconds after submitting the booking form with all the correct information.

Test case 3

1. Toggle the menu.
2. Access the history page by clicking on History.
3. Confirm that the appointment just booked appears in the history.

Now, let's examine the top-performing AI testing tools I found and see how they followed these instructions and evaluated the website using their AI features.

Katalon Studio

[Katalon Studio](#), launched by Katalon Inc. in 2015, is an automated software QA tool that supports testing for mobile applications, web apps, desktop apps, and APIs. The company's website highlights that it has “embedded AI across

our entire platform to test faster, see clearer, and streamline test automation with fewer bottlenecks.”

Katalon Studio was the first tool in Katalon’s ecosystem. Since then, two additional tools have been added. Katalon Recorder is a browser automation extension for creating and running Firefox, Edge, and Chrome tests. Katalon TestOps is a test-orchestration platform that centralizes test planning and management activities, streamlining DevOps processes and enhancing cross-team collaboration.

The AI-augmented testing features in Katalon include:

- Generating Groovy code from plain English instructions (Groovy is the scripting language used for writing test cases in Katalon)
- Automatically generating test scripts based on prompts
- A Virtual Data Analyst feature that analyzes all your TestOps data and generates reports
- Self-healing capabilities

Katalon’s self-healing AI, as noted previously, automatically helps you fix tests that break during runs. You don’t have to manually maintain existing test scripts when you ship a new feature or change a component. Regression test plans are also handled automatically: the AI engine instantly reruns your existing functional and nonfunctional tests to ensure that your software’s previously developed and tested components still perform correctly even after you’ve added new changes.

To create test cases in Katalon, you typically either record tests and playback or write test scripts with Groovy.

Practical example

In this example, I used StudioAssist AI, Katalon’s generative AI, which helps programmers write test cases from plain-language prompts. I used it to write test cases for the healthcare service website. For the sake of this test, I acted as a stakeholder who doesn’t know the Groovy syntax. I used the StudioAssist AI feature in Katalon to generate Groovy scripts, which set up my tests. I wrote the test I wanted in the prompt, and it created a test script for me in Groovy, which I then ran to evaluate the software. StudioAssist also helps explain the function of each line of code it generates.

I created a new test project, set up a test folder, and navigated to the script tab to begin writing my tests. Here is the prompt I gave StudioAssist AI:

Prompt:

I want to write a test case performing the following steps:

1. Open the browser to <https://katalon-demo-cura.herokuapp.com>
2. Click the make appointment button named 'Page_CuraHomepage/btn_MakeAppointment'
3. Fill username in the 'Page_Login/txt_Username' object with the value in the 'Username' variable
4. Fill the password in the 'Page_Login/txt_Password' object with the value in the 'Password' variable
5. Verify that the appointment div 'Page_CuraAppointment' exists within 10 seconds.
6. Close the browser

Katalon StudioAssist generated test cases written in the correct Groovy syntax (see it in full in [Example 4-1](#)) that executed the test script when it was run (see Figures [4-1](#), [4-2](#), and [4-3](#)).

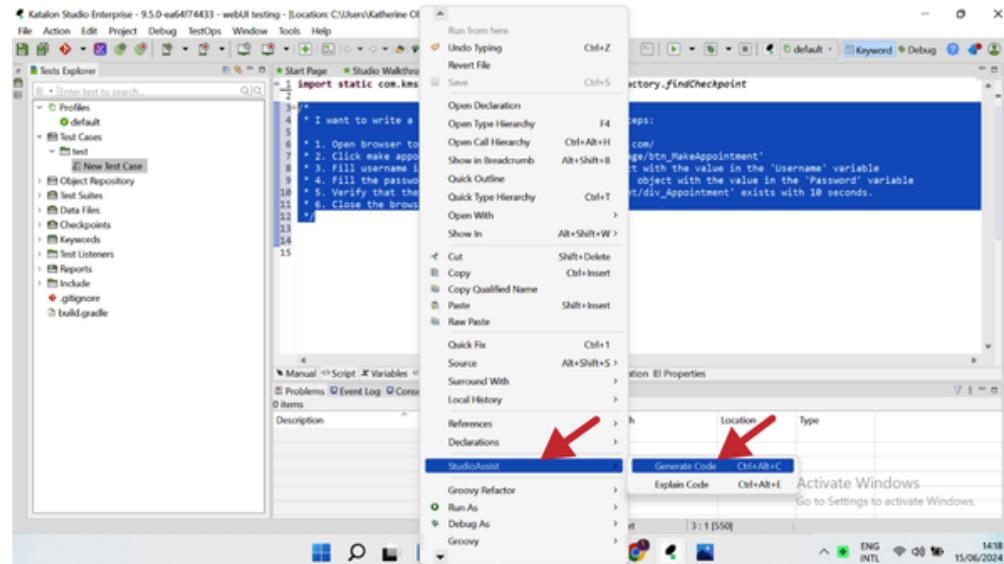


Figure 4-1. Generating tests with Katalon is intuitive when using the StudioAssist option in the UI

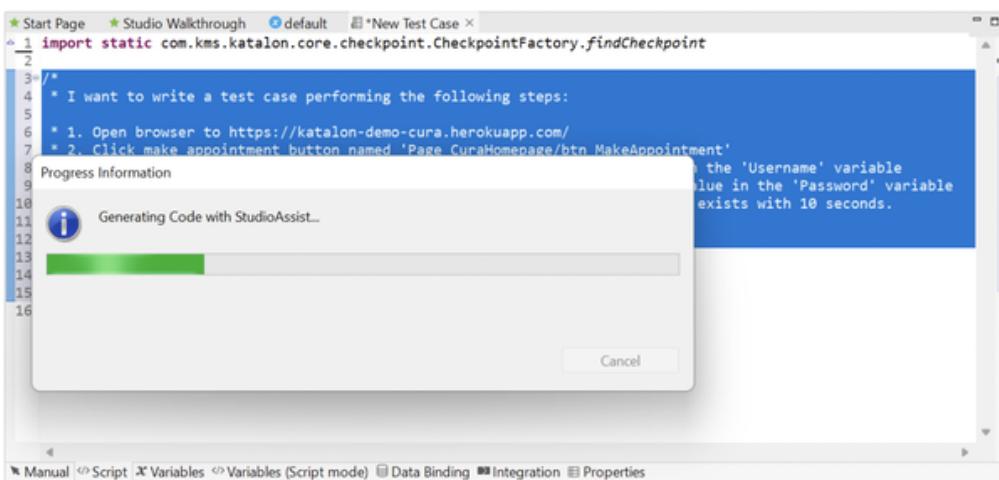


Figure 4-2. Generating tests with Katalon takes a few seconds

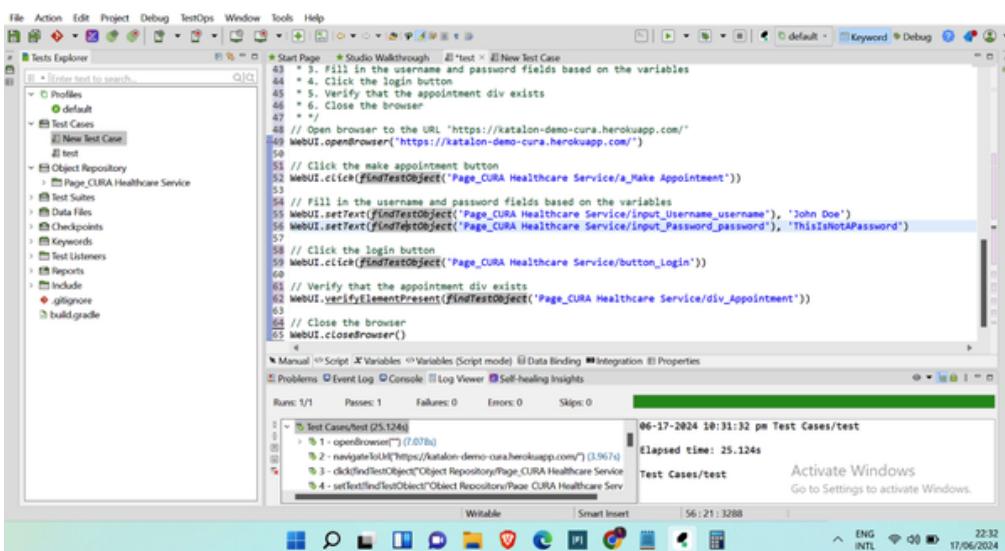


Figure 4-3. Tests generated by Katalon and executed on the StudioAssist UI

Example 4-1. Full code of the tests generated by Katalon

```

/*
 * I want to write a Katalon Studio test case to perform the following steps:
 * 1. Open browser to the URL 'https://katalon-demo-cura.herokuapp.com'
 * 2. Click the make appointment button
 * 3. Fill in the username and password fields based on the variables
 * 4. Click the login button
 * 5. Verify that the appointment div exists
 * 6. Close the browser
 */

// Open browser to the URL 'https://katalon-demo-cura.herokuapp.com'
WebUI.openBrowser('https://katalon-demo-cura.herokuapp.com')

// Click the make appointment button
WebUI.click(findTestObject('Page_CURA Healthcare Service/a_Make Appointment'))

// Fill in the username and password fields based on the variables
WebUI.setText(findTestObject('Page_CURA Healthcare Service/input_Username_username'), 'John Doe')
WebUI.setText(findTestObject('Page_CURA Healthcare Service/input_Password_password'), 'ThisIsNotAPassword')

// Click the login button
WebUI.click(findTestObject('Page_CURA Healthcare Service/button_Login'))

// Verify that the appointment div exists
WebUI.verifyElementPresent(findTestObject('Page_CURA Healthcare Service/div_Appointment'))

// Close the browser
WebUI.closeBrowser()

```

```
'ThisIsNotAPassword')

// Click the login button
WebUI.click(findTestObject('Page_CURA Healthcare Service/div_ThisIsNotAPassword'))
```



```
// Verify that the appointment div exists
WebUI.verifyElementPresent(
    findTestObject('Page_CURA Healthcare Service/div_Appointment'))
```



```
// Close the browser
WebUI.closeBrowser()
```

As you can see, the generated test fulfills the instructions I provided, and the code is written in the correct syntax.

Pros

- StudioAssist is easy for nontechnical users to use and debug, since it transforms natural-language prompts into the correct Groovy testing syntax.
- Built-in keywords and templates speed up the test-creation process and reduce the need for extensive coding.
- Its self-healing capabilities automatically update test scripts when there are changes to the application's UI.
- StudioAssist integrates with popular CI/CD tools and testing frameworks like Jenkins, Git, and Jira.

Cons

- Katalon requires you to download and install StudioAssist (shown in the preceding screenshots). This adds some additional setup work.
- Katalon can sometimes be slow, particularly when dealing with large test suites or complex test scenarios.
- There is a bit of a learning curve with the Katalon StudioAssist UI. Some options are buried inside the cascade options from the top bar, and you'll need to learn keyboard shortcuts.

I rate Katalon a 9 out of 10. It helps a lot with writing tests from plain English text prompts and executing them against the application I want to test, both within the same UI. The only reason I won't rate it 10/10 is the learning curve

pointed out in the cons list. It could certainly be more intuitive, although this is a typical UX for complex enterprise products, which Katalon already is.

Let's turn now to the second tool.

testRigor

The next tool I tested is [testRigor](#), an AI-driven automated tool designed to streamline software testing. Unlike traditional testing tools, testRigor allows developers to create and execute tests without writing code. Its NLP capabilities allow you to describe your application's functionality in plain English. The AI then generates, executes, and reports on test cases, significantly reducing the time and technical expertise required for comprehensive software testing.

Practical example

In my evaluation of testRigor, one feature that really stood out was its completely codeless test-creation process. I did not have to write a single line of test code. Instead, I provided my test site URL and a brief description of my application and how it should behave. I also provided my test goals and specified the number of test cases to generate. The AI handled everything, from generating tests to executing them to generating a detailed test report (see Figures [4-4](#) and [4-5](#)).



Figure 4-4. Prompt and description to generate test case

Figure 4-5. Tests were executed against the testing app and passed successfully

The goal of the testing, as you may recall, was to check whether a user can log in in less than 10 seconds and successfully book an appointment.

Pros

- testRigor uses Behavior-Driven Test Case Creation, which allows for the creation of tests based on how users interact with the application. This bypasses the technicalities of testing syntax, which can prevent attrition for nontechnical users or smaller teams.
- testRigor's testing product is very accessible, which makes it stand out from the crowd. It's fully cloud-based, which eliminates the need to install

additional software (unlike Katalon). This makes it easy to access and use from anywhere.

- It integrates with popular CI/CD pipelines like Jenkins and CircleCI and supports bug-tracking tools like Jira, which make it seamless to integrate with the tools that teams are already using.
- The self-healing functionality, just like Katalon's, reduces the maintenance burden on the testing team whenever existing application workflows are changed.

Cons

- Bypassing actual test writing is great for smaller teams and occasional users, but I doubt it would be practical for larger teams that already have a large testing infrastructure in place. For those software teams (which are the majority), the value of automated testing is to generate the tests in correct syntax.
- A cascade con of this bypass is that testRigor doesn't offer the same flexibility and control as traditional testing languages and frameworks. It would not work well for complex test scenarios or extensive application workflows.

Due to these limitations, I rate testRigor a 7 out of 10. Beyond that, it's a great UX that "just works," and it's a perfect fit for smaller teams that don't have a complex testing infrastructure in place already, or teams whose testing needs are occasional and who just want to check that the product is working as per the requirements.

Tool Comparison

Katalon and testRigor have strengths that cater to different testing needs, though both leverage AI and machine learning to enhance their functionalities.

[Table 4-1](#) provides a comparison.

Katalon

Katalon offers a robust suite of features designed to handle complex test scenarios. It is particularly useful for large-scale testing projects where comprehensive test coverage is critical, and for software development teams that already have a testing infrastructure, team, and

processes in place. While the learning curve is steeper than with testRigor, Katalon's depth of features and flexibility in handling diverse testing requirements make it a powerful tool for a larger number of software development teams, especially larger ones or those working on complex products.

testRigor

I was impressed with testRigor's simplicity and ease of use. The learning curve is notably short, and I found it remarkable how fast I went from signup to actual test results. This tool excels in environments where product features change frequently, requiring rapid and continuous testing. I'd say testRigor is best suited for startup teams or occasional one-off users who don't have an existing testing infrastructure in place and whose product requirements may change too often for them to even set up such a robust testing environment. On the other hand, testRigor poses limitations for those teams where Katalon excels; that is, for larger teams and more complex product workflows.

Table 4-1. AI testing tools overview

| Tool | UX | Test performance |
|-----------|------------|------------------|
| Katalon | Repository | 9/10 |
| testRigor | Browser | 7/10 |

Conclusion

Of the tools analyzed in this chapter, Katalon emerged as a good pick for larger teams and enterprise products, while testRigor proved to be a winner for startups and side products. That covers the software-development market nicely, and showcases how teams with different types of products and levels of maturity can benefit from using AI testing tools.

If you've ever worked in software testing or QA, or if you've simply written unit tests for any code you wrote, you'll know how laborious it is to write tests and keep them updated as an application evolves and gets extended.

I've often been part of conversations about budget planning and roadmap discussions where robust testing was postponed, or outdated tests were simply

framed as technical debt that should be phased out. It's very common for both technical and nontechnical stakeholders to have biases against proper testing practices, and one of the key reasons for that is how significant an investment it has been, historically, to have them.

That brings us to the bulk of the value that AI testing tools can bring to the table. In software development, we're constantly looking for occurrences of the Pareto principle: "What's the 20% of effort that will return 80% of this roadmap item's value?" As a CTO, I've been in the center of these discussions many times. In QA, the 20% of effort that creates 80% of value involves defining the application workflow properly; talking with users and clients about the issues and edge cases; going the extra mile to map out nuances for the software developers who will implement the requirements; and, ultimately, conducting user acceptance testing as a final gatekeeper before going live.

The other 80% of effort, which creates 20% of value, is the actual grunt work of writing and executing each test to verify if the code fulfills the requirements. AI tools excel at this task. Being able to provide instructions in natural language and get back tests written in proper syntax, ready to execute, is a huge time-saving use case. Having those self-healing capabilities to update tests whenever application code is changed is a great backstop for when tests become deprecated and are simply commented out, as pressing priorities emerge to get a release to production.

These are the day-to-day decisions that so often relegate proper software testing to a second-order priority. AI tools can help alleviate those concerns and contribute to ensuring that software running in production is properly tested and bug-free. This can't be done by AI tools *alone*, since these tools won't replace humans. Quite the opposite: the human tasks in QA are critical, as they define the scope of testing and serve as key guidelines for the AI tools to do the grunt work at a high quality standard.

Once again, "AI + human" is a combination that improves an often-frustrating process to produce a higher-quality output.