functionize

Test automation essentials

Why you need it and how to get started

eBook

Executive Summary

Test automation has changed the software industry. It enables CI/CD, accelerates software development release cycles, and frees up time for test engineers, who can spend their time more productively.

This ebook examines common types of software tests and their evolution alongside common software development practices. It's an introduction to testing methodologies within an evolving context of bug finding-and-fixing, including both manual and test automation. And it'll help you understand why test automation is so critical for the modern software development lifecycle.

A brief history of testing

Testing is a vital part of the software delivery lifecycle, today, but it wasn't always a separate, parallel task. Back when software systems were relatively simple, computer resources were shared and in short supply. As a result, it could take hours or days to build an application. That motivated programmers to optimize the application creation process. Developers concentrated on writing error-free code that would run as efficiently as possible. Typically, this meant writing machine code or assembler code. However, this didn't scale well, as applications needed to do more-and-more and human attention spans and time resources are limited.

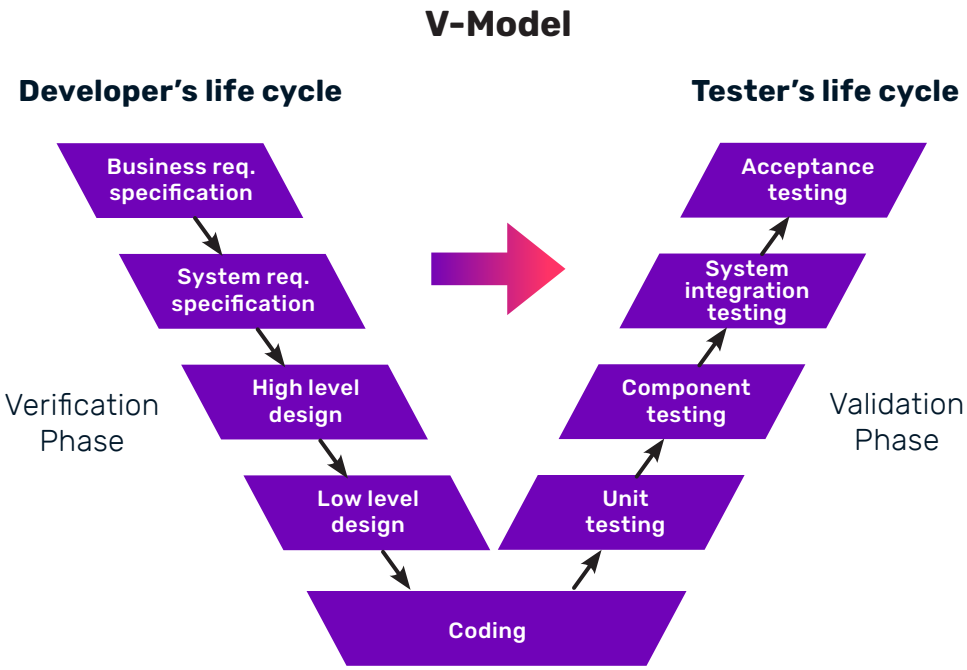
As computers became more powerful and more pervasive, developers shifted to more abstract languages, such as Simula, Smalltalk, and C. These languages allowed developers to write code more efficiently, which increased both productivity and code complexity. Ultimately, this saw software testing emerge as a discipline in its own right.

Over the past 40 years, testing has evolved significantly. It started as a manual process for skilled engineers who worked out the likely ways in which a program could fail. Then, people developed ways to automate testing. Those methods were rudimentary at first, and became more powerful over time. Nowadays, as Agile and DevOps practices become more prominent, companies are keener than ever to automate testing. This allows them to accelerate the release cycle, which lets them get new features and products to market faster.

Despite the introduction of automation 20 years ago, we still have companies approach us that primarily rely on manual testing. Hopefully, this guide can help you make progress towards sensible test automation.

The V-Model in software testing

Typically, software testing is presented as a hierarchy that matches the steps in the design process. This model, often called the V model, comes from hardware manufacturing. The following diagram shows a typical example.



This traditional testing hierarchy dates back to the days of waterfall development. However, large parts of it are still relevant to today's more agile development methodologies. Let's examine how each of these applies to today's testing environment.

Unit Testing

In *unit testing*, there is a test for each individual function. This is a form of white-box testing. That is, the developer knows exactly what the function should do and what the result ought to be.

Good unit tests check both valid and invalid outputs. For example, a unit test should learn how the function handles failures as well as how it responds to successful outcomes. One way to manage this is using *mutation testing*, in which the source code is changed in controlled ways, and you determine if your tests correctly identify the change as a failure.

Component Testing

In any system composed of several pieces – an audio/video system, a car, or a complex computer system – many components work together. Each component does one precise thing; the system results from several functions that work together. In building software, you test individual components with unit testing, but you also have to concern yourself with how they fit together.

Component testing typically also uses the *white-box* approach, but you can use *black-* or *gray-box testing*. With the latter, you make limited assumptions about how the code works.

Again, the aim is to check that the component functions correctly. To do this, the tests need to provide suitable input values. This may involve someone from the testing team, but more often, it is done by the developers.

Integration Testing

Integration testing happens once several components are completed. The goal is to ensure that the components fit together logically. For example, the inputs and outputs should be correct, and each should send the right data or messages to the other. For example, in a large airline reservation system, the luggage tracking system needs to record that a bag was taken off the plane, and also correctly send the tracking IDs to the component that reports that the suitcase was put on the conveyor belt. (Or at least to confirm that the bag was sent to Belgium by mistake, and launch the customer-support process for the traveler.)

Often, integration testing replaces some components with faked or simulated inputs. This allows teams to develop components in parallel without blocking one another.

Integration testing is usually black box. That is, the tests don't care what happens inside each component. They just care that the correct result happens.

Acceptance Testing

The highest level of the traditional testing hierarchy is *acceptance testing*, which checks that the code you produced behaves as the end-user or customer expects. The aim is to verify that the functionality achieves the business requirements specified at the start of the project. In agile development, these requirements may have evolved, so you instead check against the latest user stories.

Regression Testing

Software is developed iteratively. As you develop new features, you create new tests in parallel. That is, if you add a set of features to track airport luggage handling, you need tests to find out if the software is correctly doing its job.

You need feature tests at every level in the hierarchy. However, you also need to ensure that the existing code continues to work. There's no point in adding a new feature that breaks an old one in the software!

Regression testing is for making sure that code changes don't have an adverse effect on the application. It is essential to test every existing feature, even if there is no obvious linkage between the old and new code. This means repeating all tests every time you create a new build, which can require you to run many thousands of tests, at every stage in the testing hierarchy. In an established project, regression testing may account for the majority of the time your team spends on testing.

In practice, you can improve matters by applying common sense. Often, new features overlap existing ones, which means you can modify the existing test or replace it with a new test. Some parts of your code may be so stable that running the tests over the entire release cycle is enough.

Beyond functional testing

While the testing hierarchy above remains useful, it misses a number of essential elements in software testing. Those tests are mainly functional tests that check that the code does what it is meant to and that the application does what it's designed to do. Each functional element of the software is exercised to verify that the output is correct. This is what people typically think testing is all about.

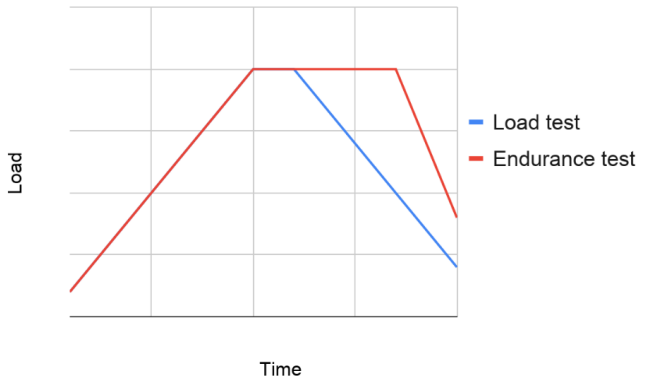
The primary challenge is to ensure you really are testing everything you need to. For instance, many applications have functionality that is only used occasionally or can only be accessed by certain users. Some functions may only run once. Other functions may only be called if things have gone wrong. All this means that functional testing requires close collaboration among test engineers, developers, and the product team.

Whatever type of testing you do, the software should fail gracefully. For instance, it should give users useful error messages, or at least redirect them to a static page apologizing that the service is slow or unavailable. Depending on your setup, it may also trigger an automatic scale-up of your backend servers.

However, an application can do its assigned functions... but not do them well. That's where other types of testing come in.

Performance testing

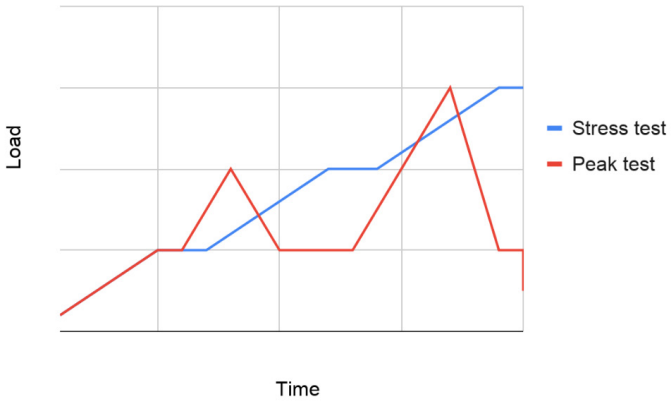
The goal of performance testing is to evaluate a system's efficiency under real-world loads. After all, it doesn't matter if the software gives right answers if it takes so long that it's no longer useful. What's the point of a super-accurate weather prediction that comes out a week late?



Load testing

One form of performance testing is load testing, which involves checking that your application can handle the expected load from users. The application shouldn't slow down excessively and certainly, it should not stop. Load testing can also encompass system monitoring, in which you track the load on backend systems.

Closely related to this is endurance testing, where you maintain the load for a protracted period.



Stress testing

Nobody wants to see software fail — but way too often, it does. It's far better to find out what the failure point is under controlled conditions, so you can measure when it's likely to occur and under what circumstances.

With ***stress testing***, you try to actually break the application. You want to see what happens when it is subjected to extreme load.

A related test is ***peak testing***, where you see how your system responds to sudden, transient traffic bursts.

Other forms of testing

Just as there are plenty of ways to create software, the computer industry has a lot of options for testing it. Many of these are related to security (such as penetration testing) or usability (such as accessibility or A/B testing). Here's some of the more common ones you may come across.

Usability testing

Key types of usability testing include:

A/B testing. Any kind of user experience (UX) design requires a comparison: Which works best? A/B testing and related techniques are about comparing different UX options to find which one is easiest for a user. Typical scenarios include:

- Testing the wording of action buttons (e.g. is "Buy Now" better than "Add to cart"?)
- Finding out which "other users bought" widget gives the best results
- Checking which layout is the most user-friendly
- The key in A/B testing is to have good instrumentation of your application. This allows you to record and analyze user interactions properly.

Beta testing. This is a widely-adopted form of acceptance testing. Before *beta testing*, you recruit willing end-users to use a pre-release version of your software. They know it may be buggy, but they see advantages in getting early access. Often it's because they are enthusiasts who welcome the chance to preview new features and appreciate the opportunity to give developers direct feedback. Often, beta testing is about both identifying bugs and refining user flows. Some features may prove unpopular and end up being removed in the final release. Other times it may allow the development staff to identify new user needs that weren't met.

Canary testing. Instead of exposing your entire user base to the new code, in *canary testing*, a small percentage (around 5-10% of users) are moved to the new code. Your aim is to compare their user experience with that of users still on the old version and identify significant problems. This way, your new code is tested on a wide range of devices, runs on production servers under real-world conditions, and exposed to all the "unexpected" behaviors that only real users can come up with. Also, by comparing the new code with the existing code you can both spot unexpected behaviors and record changes in performance.

There are plenty more, such as eye-tracking (to see what a user looks at) and heatmaps; accessility (to learn whether disabled users can access the software); and learnability studies (in which the testers find out how "intuitive" the user choices actually are).

Security testing

There's a whole host of security tests you need to do. Indeed, security testing could fill a whole ebook by itself. Here are a couple of the more common forms of security testing:

Penetration testing. The effort is to break into your own systems in order to test their security. The aim is to prove that your systems have no security vulnerabilities that might be exploited by a hacker. It is usually performed by specialists who are external contractors..

Vulnerability scanning. Where are the weak points? You need to test every application for security holes, which suggests the use of vulnerability scanning. These test suites check for SQL database errors, website security holes, and other common defects. It may identify network and system weaknesses and suggest solutions.

Risk assessment. Not all applications face equal security risks. A simple smartphone game needs less protection than does an online banking application. The only way to establish the risks faced by your application is to complete a proper security risk assessment. This should be done according to a standard, such as ISO 27001 or NIST SP 800-30; it's usually performed by specialists who are trained to interpret the results.

Those are just the start. Testers and developers who delve into applications' vulnerability are also apt to use tools for testing data storage; compliance with privacy requirements; and database security weaknesses.

Testing, at its human limits

As you can see, a lot of work is involved in testing any software. Getting it right requires knowledge, experience, and, above all, time. The more complex your software gets, the harder it becomes to ensure code quality and to create happy users. More code means you need to run more tests. Those tests get more complicated, so they take longer to complete. At the same time, your software becomes more complex, so you need to develop more advanced tests. This means you need more manpower and computing resources for your testing. Ultimately, you can end up with testing requiring more time than you can afford. The solution: test automation.

Introducing test automation

Originally, all software testing was done manually. However, that approach doesn't scale very well. Manual testing can only proceed at the speed of the test engineers. A typical engineer spends around 40-60 hours a week at work. A manual test can easily take several hours to complete, especially if it needs to be done on multiple devices. So, a single

engineer might manage to run 10-15 tests in a given week. However, test suites can have thousands of tests. You would need to employ dozens of engineers to prevent testing from becoming a roadblock.

The solution is to automate as much of the testing as possible.

About 20 years ago, developers began looking at ways to automate aspects of the testing process. Unit tests were easy to automate. However, integration tests and regression tests were much harder to tackle.

Unit testing

Unit tests are functions and classes that developers add to the code in order to test the functionality. For instance, your function might calculate sales tax for shopping transactions. Here. the unit test needs to check that the dollar amount is correctly calculated. These classes can be called manually, but more often the developer creates a helper class to make it easier.

Automating unit testing is relatively easy -- compared to other tasks, anyway. Each test is just a function or class. So, it is simple for a developer to write a script that calls each test regularly. Then the tests can be called automatically each time developers add new code to the master project.

Typically, this is done for you by continuous integration tools. These tools generally won't accept a new pull request until they check that the code passes all unit tests. Some project managers even impose rules on test coverage, such as the percentage of the code that is being tested.

User interface testing

User interface (UI) testing is an important element of test automation because you can perform all functional and regression tests on your system through the UI.

There are broadly three forms of automated UI testing: scripted tests, Intelligent recorders, and AI-powered test tools. They all attempt to automate the manual test plans used in functional and regression testing.

UI testing requires you to create detailed test plans for each user flow you need to test. A test plan is a list of steps that the computer can follow to walk it through the flow. Steps include navigating to a particular page, selecting elements, interacting with the UI (clicking, entering text, etc.), and verifying that the correct page has loaded. These plans may be automatically created by a test management tool or manually in collaboration with the product team.

Back end and API testing

Typically, a modern application consists of a front end (the part people interact with, such as the actual "app" on a smartphone or a website user interface) and a back end, where the data is stored and processed.

These are connected via an application programming interface (API). These back end systems need to be tested for functionality, stability, and scalability. This is typically done using API testing and performance testing.

There are various approaches to automate this part of the software testing process. You can design scripts to exercise the APIs the application uses, in a similar manner to unit tests. Performance testing can be automated using scripts that stress the APIs with large numbers of simultaneous API calls. Specialist tools can help, too, such as **Artillery** or **Gatling.io**.

Why is test automation essential?

Test automation has transformed the world of software development. Arguably, it ranks alongside cloud computing as an enabler of the explosion in mobile and web applications over the past decade. Among the reasons why:

Scale: Automating tests transforms the number of tests you can run. Computers can run tests 24 hours a day, 7 days a week. The upshot is, you can run many more tests with the same staff.

Complexity: Test engineers, freed from the time-consuming tedium of manual tests, can spend their time on other things. Instead of clicking through the same-old-things, the testers can create tests for more complex scenarios and enable more powerful applications. The effect of this is apparent if you compare the earliest apps released for the original iPhone against today's apps.

Speed: Naturally, automating tests means they can be completed faster. The faster your software passes testing, the faster you can release the application.

Expect the primary gains from test automation in regression testing, where established tests (that you know actually exercise the functionality) can be done and re-done as often as possible.

When to avoid test automation

There are a few situations where test automation isn't suitable. Among them:

Unpredictable outcomes. Some applications have unpredictable or varying outcomes from user interactions. In games, for example, often an intelligent game engine decides the outcome of everything a user does. In such cases, the result of any test will be unpredictable. If you can't predict the outcome, you can't tell the test how to distinguish a pass from a fail.

Recreating reported bugs. Users often report defects that are hard for developers and testers to recreate. While you attempt to recreate the bug, you need to do exploratory testing -- which means manual poking-around. Automated testing struggles to help in this situation. All that you can do is check if the bug happens during a known or planned user journey.

Testing complex and dynamic UIs. Some UIs are too complex to be tested automatically. You can test the planned user journeys, but not the unplanned ones. Users may well do unexpected things when they use your UI. Modern UIs offer so many paths to navigate that it is impossible to create tests for all these.

Likewise, modern UIs are often dynamic; the UI is generated uniquely for each user. The user may even be able to customize the UI using widgets or other tools. In such a situation, the tester is faced with non-linear flows that materialize as a result of the user's (past or current) behavior.

Scripted test automation, with an emphasis on Selenium

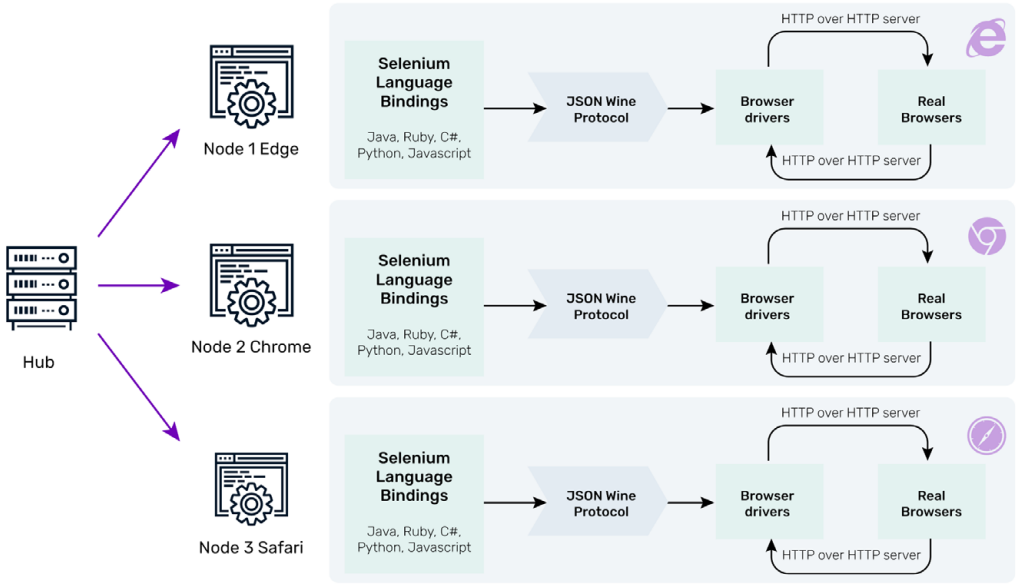
The earliest forms of test automation were developed in-house by large software companies. These were generally bespoke solutions designed to help with testing large monolithic applications. As with most custom software, it was precisely tuned; but that also meant any organizational change broke the software, and the developers had to expend time maintaining the tools rather than working on the real production code (the stuff that paid the company's bills).

The first universal framework for test automation was Selenium. Jason Huggins, one of the fathers of test automation, created the first version of Selenium in 2004 while he was working for Thoughtworks. Selenium was specifically designed to help script and automate the testing of web applications. It quickly became an open-source project and additional tools were added to the framework over time. Since then, Selenium has gone on to become one of the most, if not the most, widely-adopted software testing frameworks globally.

The core elements of Selenium are:

- **Selenese**, a domain-specific language, to define test cases.
- **Selenium IDE,** a Firefox plugin that allows you to record test cases and test suites and then replay them (but only in Firefox)
- **Selenium WebDriver**, which coordinates the replay of tests to the browser and can work across many different browsers.
- **Selenium Remote Control**, a tool to allow you to create automated UI tests in any language and run them remotely against any HTTP website on pretty much any modern browser.
- **Selenium Grid**, which allows Selenium test suites to be coordinated and run across multiple servers in parallel.

The development and integration of Selenium is coordinated by Selenium HQ. In the following diagram, you can see how its elements fit together.



vSelenium allows you to write tests in many different languages. These tests are then converted into Selenium test scripts that are passed to the appropriate Web Driver.

How Selenium works

Selenium does its best to replicate the actions of a real user interacting with a software UI. It accomplishes this using a combination of element selectors, which find particular objects on the UI, and actions, which include clicking, dragging, and entering text. It then checks the actual outcome against what you told Selenium should happen.

As a simple example, imagine a test for logging into the Facebook homepage. You need to tell Selenium to find the email and password fields, enter the appropriate information, and submit this. Selenium verifies that the page changes to the user's homepage.

Element selectors

Selenium uses several ways to select elements, most commonly ID, name, CSS selectors, and XPaths.

ID or name: In well-designed UIs, every single element on the page has an ID (hand coded or generated by a web framework) and each ID is unique. In practice, this often is not the case. Alternatively, site and app design can use an element name. Many elements will have a name, particularly elements like buttons that the user can interact with.

CSS selectors: CSS uses CSS selectors to determine which style to apply to the element. Selenium uses them back-to-front to find the element that matches that selector. You can find more about these in our blog post about **CSS Selectors**.

XPath: XPath Query Language (XPath) is a way to query (select) elements within an XML document. Because HTML is a form of XML, you can use XPath queries to select elements on a webpage. The advantage of XPath is that it allows you to select elements across different Domain Object Models (DOMs). However, XPath queries are hard to create.

Actions

Selenium allows you to perform a number of actions such as clicking, hovering, entering text, or simply verifying that an element exists. Listing all possible actions would take far too long. But let's look at some of the most common.

`click()` replicates a user clicking on an element such as a button. There are many variants includin `doubleClick()`, `clickAndHold()`, and `contextClick()`.

`dragAndDrop()` replicates a user clicking and dragging an element around the screen.

`sendKeys()` enters text within a field, such as a password box or an address details form.

How about an example?

The following is a simple example of a Python Selenium script for logging into the Facebook homepage. You need to identify element selectors for three elements: the email field, the password field, and the login button. Ideally, as the tester writing the script, you need a selector that is unique on the page and won't change each time the UI reloads. (For more details on CSS selectors, you can read our **recent blog post**.) Fortunately, the Facebook login page has simple selectors: email, pass, and loginbutton.

```
#first, import the required modules.
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

# next you need to navigate to the test URL
browser.get('https://www.facebook.com/')

# now find the email field and enter your test data
user = browser.find_elements_by_css_selector("input[name=email]")
user[0].send_keys('test@testing.com')

# find the password field and enter the correct password

pass = browser.find_elements_by_css_selector("input[name=pass]")
pass[0].send_keys('Pass12345')

# finally, find the login button and click it. NB there are a few possible selectors
you could choose.

login = browser.find_elements_by_css_selector("input[name=loginbutton]")
login[0].click()
```

That's all you need for the script. Obviously, to make this into a proper test, you need some verifications step and actions to take if the login fails.

Issues

Selenium is extremely powerful and allows you to create complex tests. However, it has several well-known issues.

Test creation

Each Selenium test script is a miniature software development project. Creating a new script is iterative and slow, requiring frequent rounds of testing and debugging. Even a

skilled developer-in-test might take a day to create and test a good script. When you add in the requirement for cross-browser testing, this gets even longer. In other words: It takes time.

The end result: Few companies manage to automate more than a fraction of their regression tests.

Test execution

Selenium still runs as a single thread on a local server. This brings a whole host of SysAdmin pain along with the costs of maintaining local infrastructure. Selenium Grid is an attempt to allow multiple servers to be managed centrally, but it's still a far cry from the scalability of a typical cloud application. Some companies do offer Selenium instances that run in the cloud. These remove the burden of maintenance, but that comes at a cost.

Test analysis

One key part of Selenium scripts is programming in the checks that verify whether the test has passed. The problem is, this has to be done manually. Realistically, you can probably verify at best 10% of the UI at each step in the script. This means there is a potential for some errors to slip through unnoticed.

Test maintenance

Every time your application's UI changes, you risk the possibility of your tests failing. If the developers change the site text from "Enter user name" to "Enter your name," the element changes and the test script fails. If the actual selector changes, then the test probably just fail at that point. But if the order of selectors changes, you may end up choosing the wrong element, in which case you may not find out until a failure occurs many steps later. That's the ingredients to a very long, cranky weekend of debugging.

Test recorders

As mentioned above, it takes a while for a skilled developer to write a test script from scratch. One way around it is to use test recorders, which capture user interactions and create a ready-to-edit script. As the name suggests, it records keystrokes and mouse movements, and converts the steps into a Selenium script.

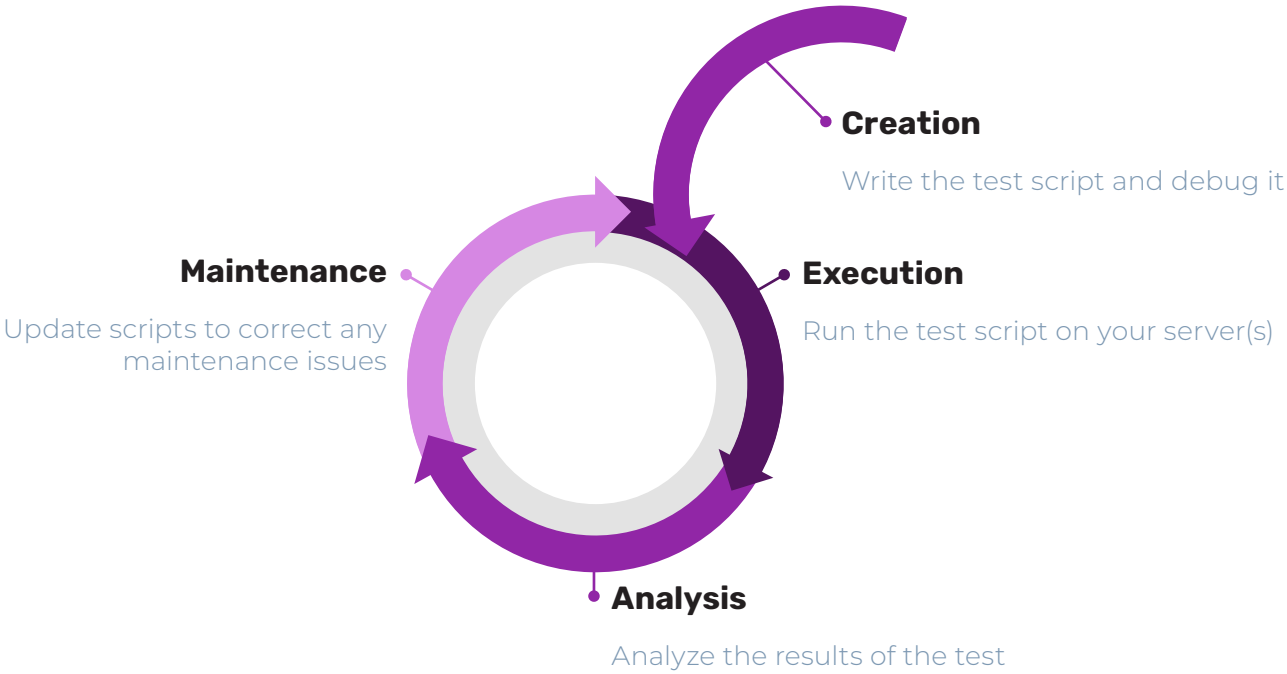
Test recorders can only help you create the skeleton of a test script. More complex functionality has to be added by hand. Moreover, test recorders tend to set CSS selectors in an unintelligent manner. This means the scripts they produce are even more susceptible to test maintenance issues.

The Artificial Intelligence Era

It is only relatively recently that computers have become powerful enough to make a difference in computing and for artificial intelligence (AI) to be more than a buzzword or a science fiction trope. While the definition of AI has changed (often at the behest of vendors who wanted a handy buzzword), most true examples of artificial intelligence are defined as narrow intelligence. That is, they are very good at a specific, narrow task.

Machine learning

Machine learning (ML) is the basis for most AI systems. Put simply, it's the process of creating computer programs that can learn to perform a task without needing to be explicitly programmed to do that task. In Machine learning, a computer learns to recognize certain patterns, and then uses pattern recognition to trigger appropriate actions.



Machine learning comes in three forms: supervised learning, unsupervised learning, and reinforcement learning. In *supervised learning*, the machine uses a large set of labeled known data. In *unsupervised learning*, the computer looks for interesting patterns in the data and seeks to draw useful, actionable conclusions. In *reinforcement learning*, the computer is rewarded every time it makes a correct decision.

Deep learning is a subset of machine learning that combines elements of all three types. It uses approaches such as artificial neural networks to process large volumes of data, and to identify patterns within the data. This is what has allowed computers to become so good at the game of Go that they can beat any human Go Master.

Computer vision

Computers can learn to recognize patterns in still or moving images, then interpret what the image shows. This consists of several stages. First, the image needs to be segmented aka object localization. That means the computer needs to identify which pixels are related to each other. Next, the computer tries to identify what each object is (object classification). Finally, the computer divides the image into semantically meaningful pieces. The end result is a system that can identify objects in a picture and recognize how they relate to each other.

Natural language processing

One of the core languages behind virtual assistants such as Apple Siri and Amazon Alexa is *natural language processing* (NLP). the field of teaching computers to understand human speech. There are several challenges with NLP, from the complexity of human language, to the many layers of meaning. Every human language has several ways to say each thing. (For example, these sentences all have the same meaning: “Your dinner is ready,” “Supper is on the table,” “Come and eat!”)

Essentially, NLP breaks down sentences into grammatical parts and evaluates how these relate to each other. The system compares this data with its grammatical knowledge to parse the meaning in the sentence.

Intelligent recorders

Circa 2015, AI and ML reached the stage where they could augment traditional test recorders. These new recorders can create robust models of each element you select in the test. They can do a better job at coping with the UI changes that could break Selenium scripts. Some of these tools go further, with drag-and-drop interfaces to help you build complex test scripts that reuse snippets you captured using the point and click recorder.

So, how do intelligent recorders compare with traditional manual scripting?

Test creation: Intelligent test recorders are faster at creating tests, and the resulting tests are also cross-browser capable. The tests are stable and don’t need manual debugging. This overcomes one of the big issues with traditional recorders.

Test execution: There aren’t any particular benefits in test execution -- but then again, there are no downsides either. Intelligent test recorders still produce Selenium scripts. So, the script itself still has to be executed like any other test script.

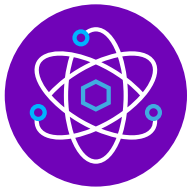
Test analysis: Again, since the tests themselves are Selenium scripts, you don't get direct benefits here. However, an intelligent recorder makes it easier to record verification steps. As a result, the tests can be more thorough than ones that are written manually. This can be important if you have a site where the result of an action is a set of relatively subtle changes in the UI, such as showing prices with sales tax for a different State.

Test maintenance: Intelligent test recorders can save a lot of maintenance time because they use better selectors that are based on ML models of the UI. So, if a button moves on the page or is restyled, the script will find it. However, other changes can still break scripts. In this case, recorders again save time by making it easy to update any steps that now contain errors. The best intelligent recorders allow you to update all other tests that use that step. In other words, if the application's webpage changes "enter name" to "enter first name" the recorder can offer the tester the opportunity to change the text throughout the test suite.

Intelligent test agents

Obviously, test automation has room for improvement, and machine learning can be a part of that. Intelligent test agents (ITAs) combine multiple AI types to create systems that can simplify the process of creating and maintaining automated tests.

Test creation



An ITA uses a combination of intelligent recorders and NLP to simplify test creation. This can be as simple as improving test recorder performance by adding an NLP layer, which allows the system to "read" and understand the UI in the same manner that a human does (but with fewer coffee breaks). ITAs can create ML models for what the test is actually doing.

At the other end of the scale are systems that start with test plans and convert the plans into functional tests. These combine NLP with unsupervised learning to model the UI and create the test. The resulting tests can work cross-browser because the test-creation process can be applied to every browser type. They can cope with complexities such as dynamic content and embedded third party widgets, such as PayPal "Buy now" buttons or CRM forms.

Test execution

Most ITA systems operate completely in the cloud, so they can access the powerful computing resources to drive AI algorithms. The benefit is that such ITA systems can access other cloud benefits, such as scalable and efficient test execution. As a result, test runs can be completed far faster than traditional approaches could manage.

A secondary benefit is that all historical test results can be stored and analyzed. This can be particularly useful when a regression occurs, such as when an existing bug reappears after it theoretically was fixed.

Test analysis

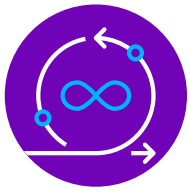
Visual test analysis uses screenshots to check a test's outcome. When the test is first executed, a screenshot is captured, showing the correct result. Later test runs compare their results against this screenshot.

The simplest approach is a pixel-by-pixel comparison. But that isn't very effective, because most UIs are dynamically generated by the browser and can introduce minute differences on the screen. More intelligent approaches use computer vision to significantly improve test analysis.

Rather than just compare images, the ITA system performs localization, classification, and semantic segmentation to identify the UI's discrete elements, such as buttons, menu entries, graphics, and blocks of text. Doing so makes the screenshot comparisons far more accurate.

For instance, the ITA recognizes that a date field changed. If the UI is powered by live data, the ITA notices that an element may have changed (e.g. prices on a shopping website). Finally, it can cope with at least some user personalization, such as where users choose to display dates in U.S. (mm/dd/yy) or European (dd/mm/yy) format.

Test maintenance



The final place ITAs really transform testing is in test maintenance. They take the benefits of intelligent test recorders and improve on them. For instance, the ITA model of the UI may include semantic information that recognizes that "Add to basket" is the same as "Purchase" or "Add to cart." They can understand how UI elements relate to each other. The end result: The system can cope with most UI changes and updates.

Functionize's approach

Functionize began life as an intelligent test recorder. However, over the past two years, our system has evolved into an intelligent test agent. This is thanks to several industry-leading technologies we introduced.

Adaptive Language Processing™

ALP™ is our NLP-powered test creation engine. ALP can understand test plans written in structured text. The testing team can sign off on the plans — which break the test down into its precise steps — and our test creation engine turns it into a set of efficient scripts that get the job done. You can edit these scripts for unique situations, if you like, but in most cases they take care of the task with little human oversight.

However, ALP can also understand more unstructured test plans. This means you can provide the Functionize tool to a user journey written by your product team and the software will convert it to a test. Because human language is ambiguous, we augmented this functionality with the ability to play back the resulting test and talk to your computer to issue corrections or changes.

Testing in the cloud

Our test cloud uses a serverless architecture to create a hugely scalable test infrastructure, which is tuned to execute thousands of tests at once. The flexible architecture is based on running each test within its own virtual container.

Thanks to nested virtualization, we can even run mobile emulators. This allows the tests to be run on mobile as well as desktop browsers.

Finally, the test cloud allows you to perform realistic load and stress testing across multiple geographic locations. Because each test runs in its own container, the test sessions are correctly handled by your load balancers – which is essential for accurate testing.

Visual testing

Our visual testing system uses screenshots taken before, during, and after each test step. It compares these against all previous test runs. Any anomalies are identified and highlighted on the screen. The system ignores elements that are expected to change, and can cope with UI design changes (e.g. styling changes). If you tell the system to ignore something, it will update its models; in future runs, it won't flag that as a failure.

Adaptive Event Analysis™

AEA™, the ML engine that drives our entire test system, combines multiple forms of ML using a technique known as boosting. This creates complex and robust models of an application's UI. It also uses NLP, computer vision, and additional classic data science techniques. The result is tests that are extremely robust to UI changes. Our results show we can slash maintenance time by over 80%.

The future

We expect intelligent test systems to continue to evolve significantly — and naturally, Functionize expects to be at the forefront of these advances. Future testing systems will be able to create far more complex tests, which are automatically updated based on monitoring how real users interact with your system. We may even see systems that are capable of simple exploratory testing.

Here at Functionize, we are concentrating on expanding the capabilities of our system. We have always adopted hybrid approaches and we will continue to do so. In the near term, this will include adding dynamic test updates as they run.

As you can see, Functionize is motivated to transform test automation and make it accessible to everyone. Won't you join us on this journey?