testim

AI Based Test Automation Explained

Autonomous Testing: Differentiating Fact from Fiction

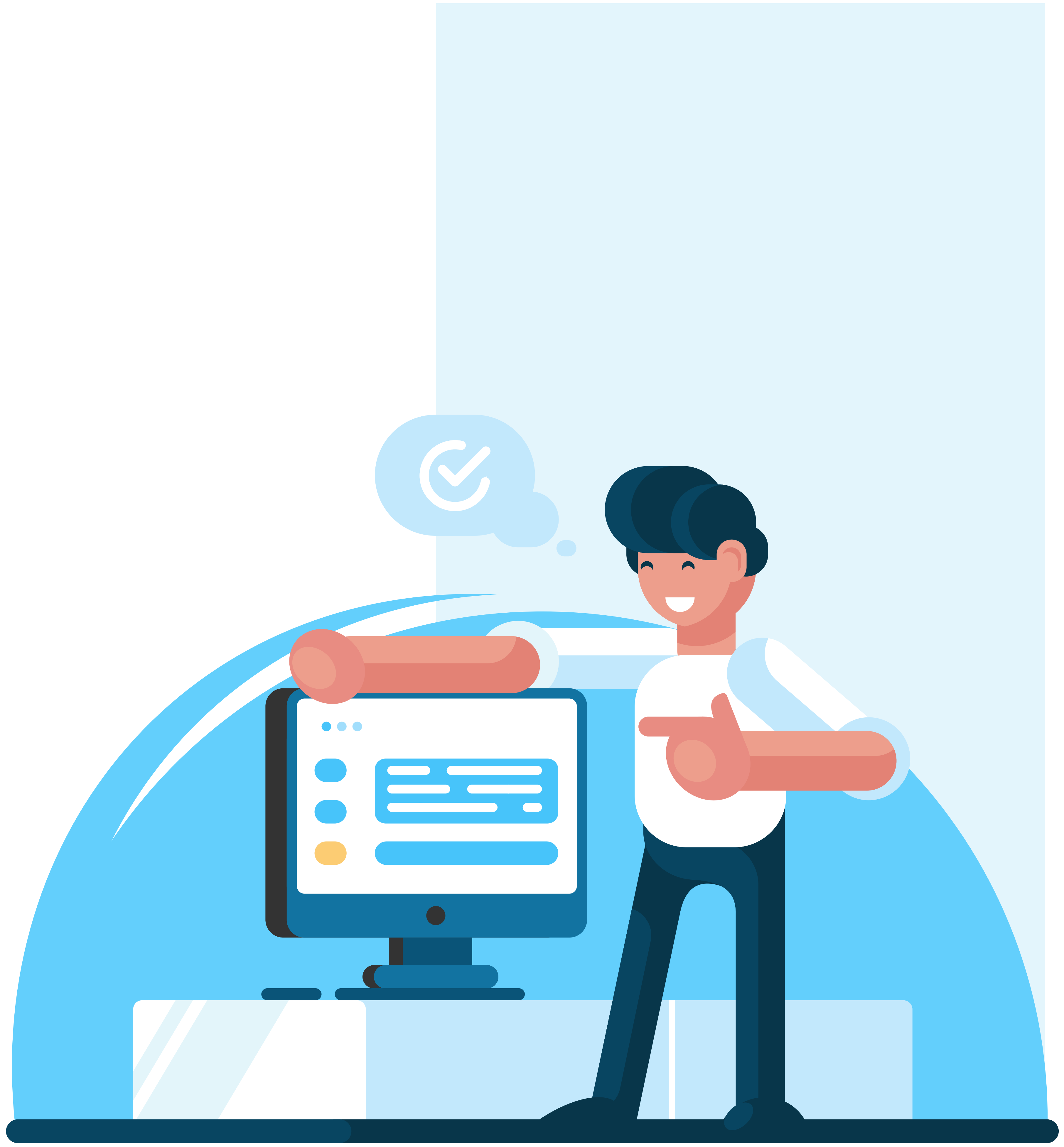
Table of **Contents**

Introduction

Similar to the concept of self-driving cars, autonomous testing refers to the use of software to aid dev teams in testing applications. With respect to test automation, autonomous testing is applicable in the areas of risk management, maintenance, and test authoring. By streamlining and automating quality assurance, autonomous testing is revolutionizing the software development process, enabling engineering teams to deliver products much faster to clients. With autonomous testing capabilities, the velocity of dev teams is only limited by how fast they can write code.

While the first of the Tesla models wasn't self-driving, subsequent releases (powered by AI) significantly improved on the user experience, gradually bringing the product closer to becoming truly autonomous. The same will hold true for autonomous software testing.

This whitepaper discusses how AI can help resolve the tradeoffs between delivering a flawless user experience and achieving faster release velocity, with emphasis on functional E2E testing.



E2E testing: A nightmare for most dev teams

A 2018 StackOverflow survey indicates that 48.2% of developers say they are full-stack while 37.8% claim to be front end developers.

The rise in the number of front end developers shows that key business logic seems to reside more and more on an application's front end, thus increasing the complexity of testing this component before release.

As such, functional end-to-end testing has become one of the biggest challenges to software quality — as evidenced by the rise in front end development.
Here's why:

Test authoring is slow

Authoring an effective, stable test takes nearly as much time as developing the feature being tested. This increases the overall time spent on quality assurance, thus increasing time to market.

Test maintenance sucks

In most software development organizations, over 30% of a tester's workload is focused on maintaining flakey tests. For instance, each time the UI changes, tests may fail and need fixing. This makes test maintenance a tedious and highly-repetitive task.

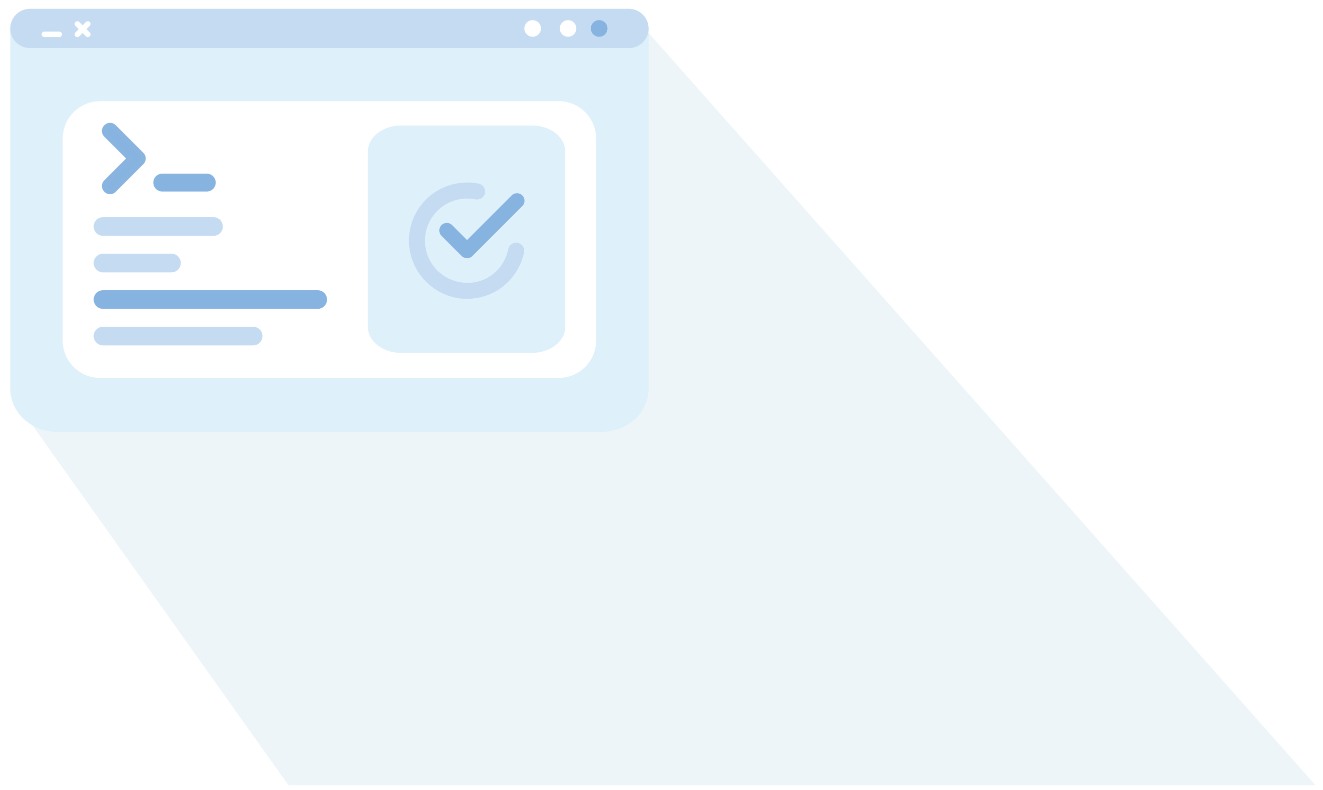
Skill set

In today's development environment, it's easier to find full stack developers than automation engineers. Coupled with the fact that the typical developer doesn't like to spend a lot of time on testing activities and manual testers find coding challenging and we have a gap in skill sets needed for effective E2E testing.

How AI can help resolve E2E testing challenges

The perfect solution to these challenges is a fully autonomous testing environment where tests are diagnosed, authored and maintained automatically. However, test automation hasn't gotten to that point...yet. Nevertheless, AI has made it possible to speed up authoring time while significantly reducing the time it takes to maintain and repair tests.

Before explaining how AI can help resolve the functional E2E nightmare, we'll take a look at the different levels of assistance in testing activities and how they can get us closer to true autonomous testing.



Level 0

Manual Authoring

Engineering teams author and maintain tests.

Level 1

Teach by example

Machines help increase the speed of test authoring while improving test stability.

Level 2

Semi-Autonomous

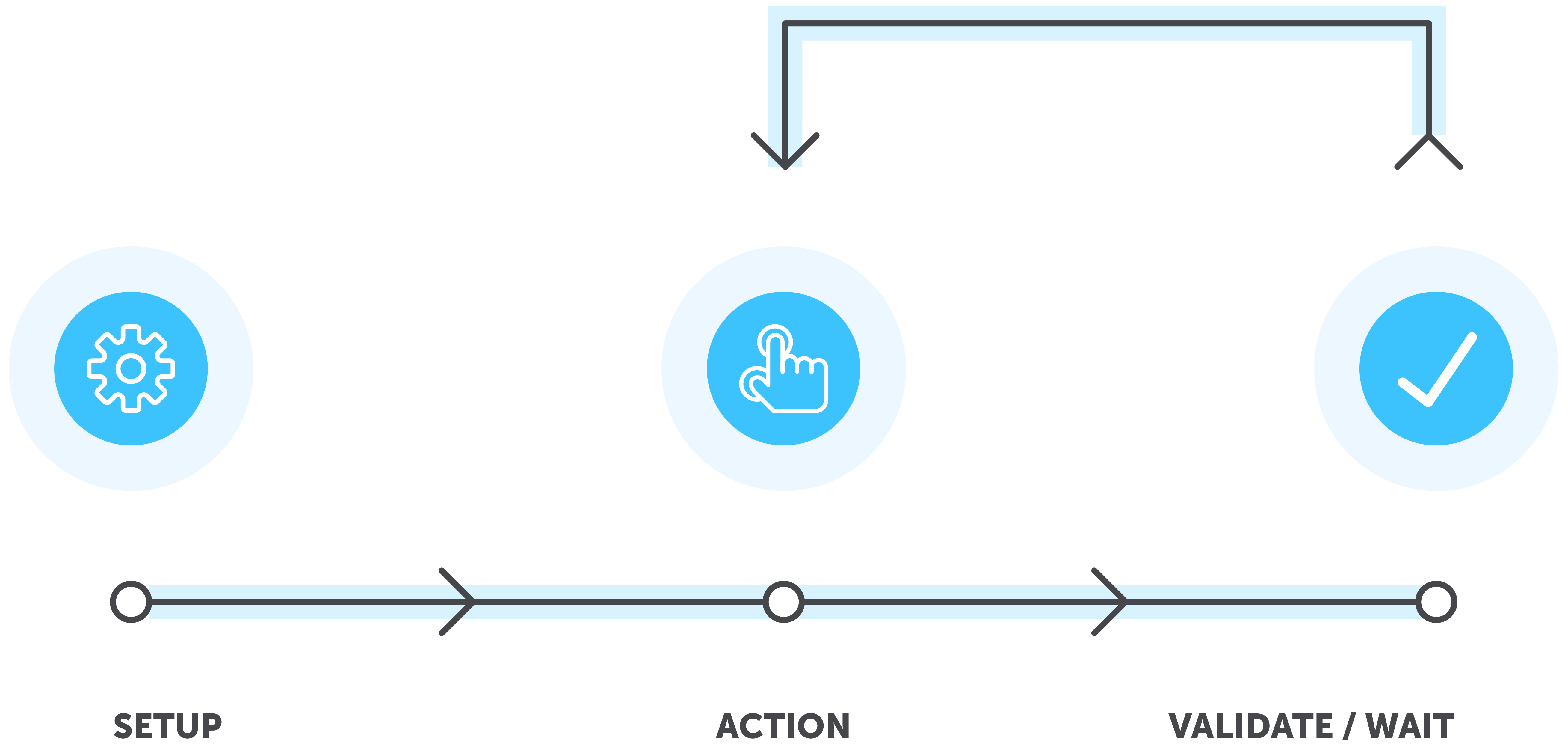
While some human assistance is required, machines write and maintain tests by observing how humans interact with apps.

Level 3

Fully-Autonomous

Machines automatically and autonomously build and maintain tests.

How are these levels applicable to functional E2E testing? The following graphic shows the elements that make up E2E tests.



We'll now review each level.

Action

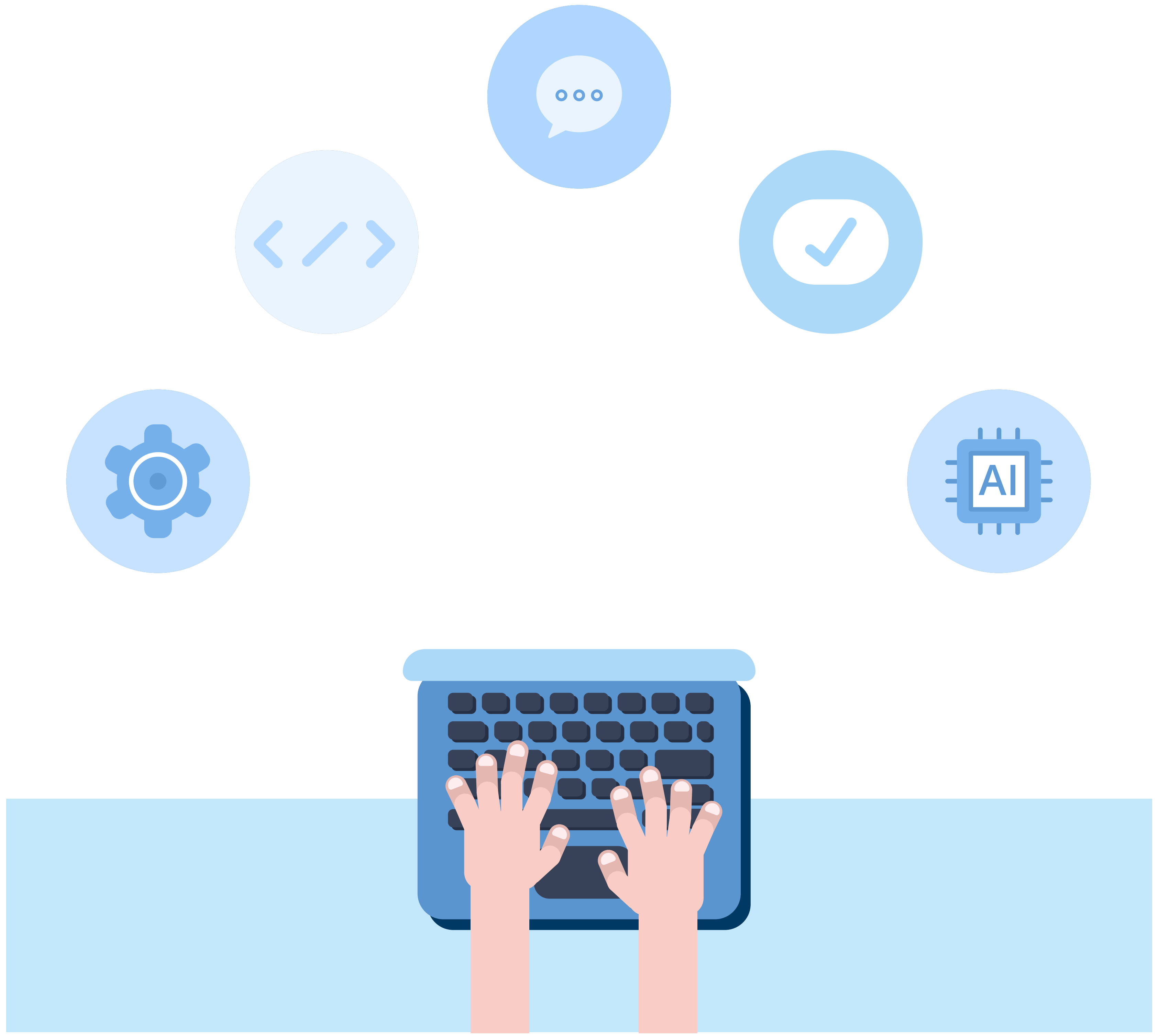
Most E2E tests consists of a series of actions on the UI. The action could be entering text, scrolling, finger taps (for mobile) or a click (for web apps).

Manual Authoring (Level 0)

Level 0 refers to a typical development environment where there's no automated assistance during test authoring. Engineering teams write everything from scratch and also have to maintain tests with every modification of the UI.

To do this, it's assumed that humans must review the entire source code (locators inclusive). There were query languages (such as XPath and CSS-Selector) that could be used to identify the position of elements on an app's UI. However, these languages could only focus on a fixed number of UI properties, making them too rigid to handle further modifications to the code. We'll refer to such languages as static locators.

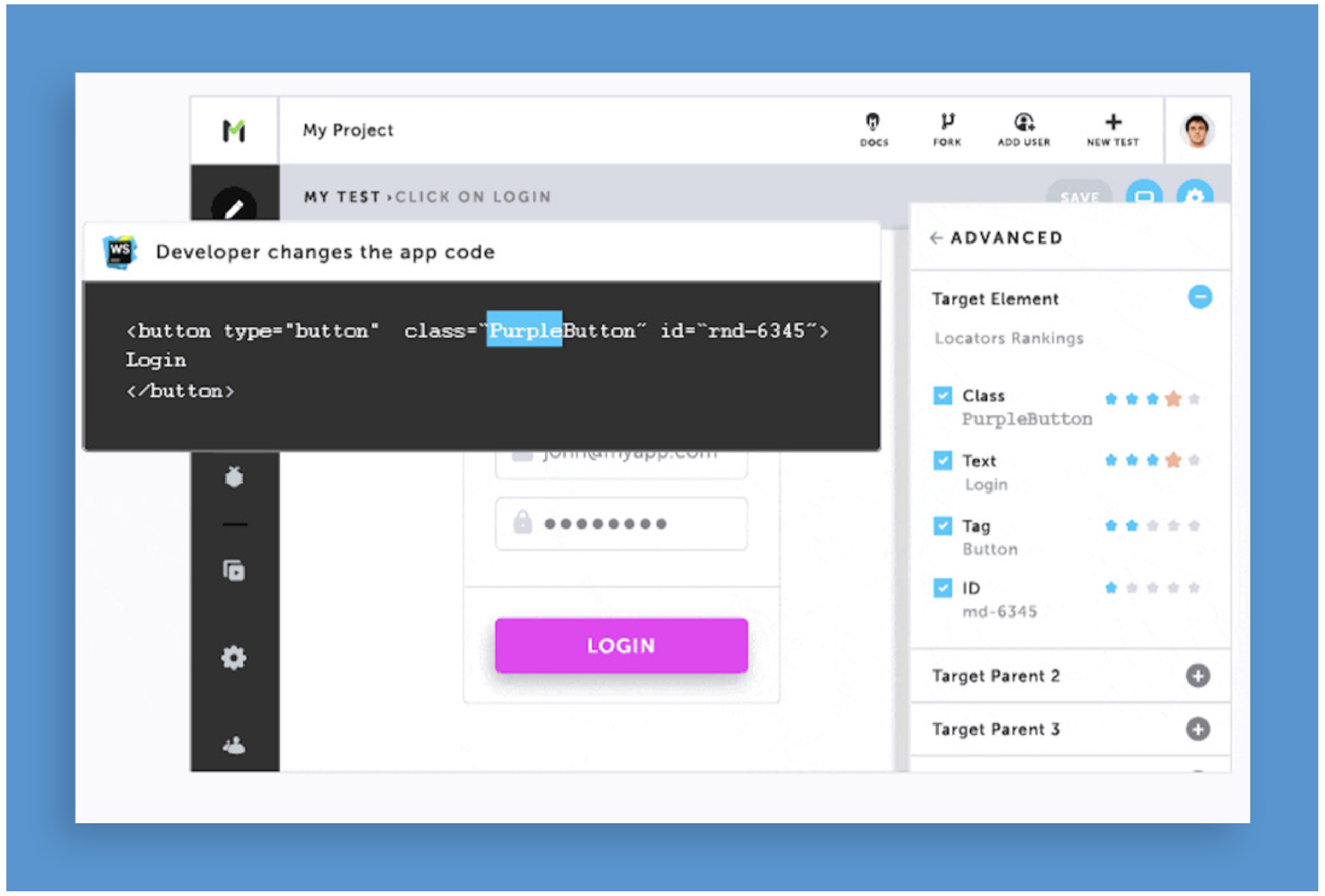
Over time, there have been several efforts to record & play back user interactions. However, these attempts eventually failed due to the limitations of the static locators being used at the time. The results were much better if developers wrote the tests since they had more knowledge of the app's source code and could leverage that knowledge to write better selectors. Also, there was improved maintenance with the underlying technology since no reuse were generated.



Teach by example (Level 1)

By moving away from the concept of static locators, we can innovate new and better locator strategies and allow machines to make final decisions. Artificial intelligence can handle large numbers of locators, and assign scores to them based on stability and quality, enabling prioritization. Besides the ability to handle a large volume of UI properties, AI is impartial and not biased.
One of the attributes that makes AI so successful at authoring tests is its ability to learn from previous executions. This means that your tests become incrementally better and stable after each iteration.

After executing millions (possibly billions) of tests across hundreds of companies and thousands of use cases, the team at Testim.io was able to put together new locator strategies that are applicable to all use cases and project scenarios. Whether you're using Vue.js, React.js, Angular.js, or other frameworks, you can leverage our tool to get much better results. Our tool is based on in-depth knowledge and practical experience gained from running hundreds of millions of iterations. This means we know which parameters are ideal and indispensable (and those that are not) for your use case.



```
<button type="button"  class="PurpleButton" id="rnd-6345">
Login
</button>
```

Manual testers are able to write E2E tests and more importantly, they only need to train the machine once. The machine introspects the entire DOM to extract attributes and generate locators by following every action a tester makes. Although this doesn't require coding, users must have a basic knowledge of engineering since they need to reuse and maybe input parameters if values vary with each call.

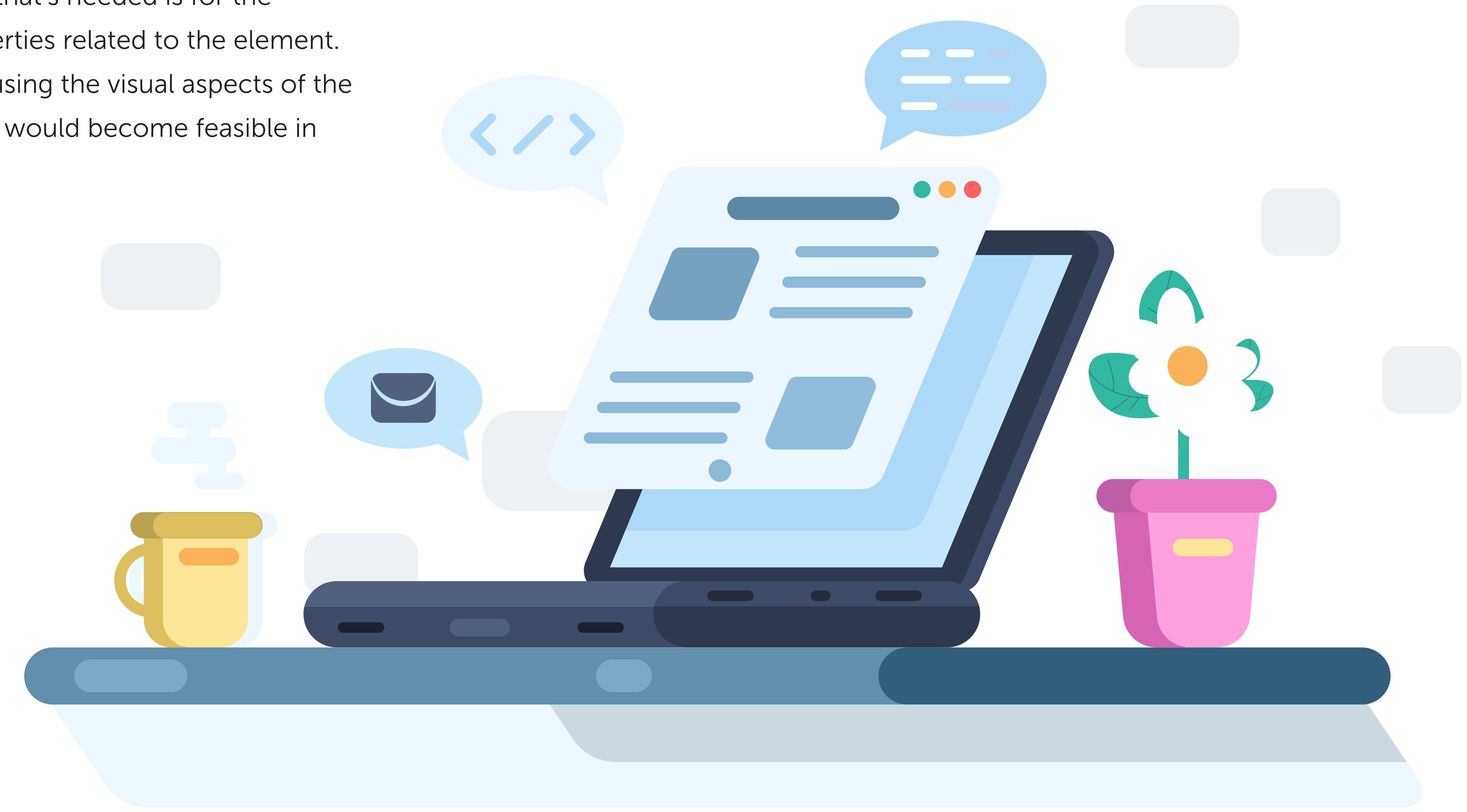
BDD (Behavior Driven Development) Support

This was created to facilitate the shifting left paradigm, where tests are executed as far to the left as possible. This led to the testing of product specs by various teams including QA and Development. While such tests don't occur often, teams sometimes use tools (such as Cucumber) to write product specifications in human-readable language and then leverage another platform that automates UI (such as Selenium) to translate those sentences into actions.

In such instances, AI can be used to automatically locate the elements. It does this by reviewing the properties of each UI element and attempting to deduce the element that fits best. For instance, extract "set userid to Joe" and look for an input with properties that's similar to "userid" and set its value to "Joe." However, this would require lots of human interaction for large apps with a high volume of elements on the screen.

Visual Driven Development

While the concept of authoring tests using the visual design mocks of an app isn’t new, it was viewed as unstable considering the fact that pixels are just one of many properties and are very fragile. With an AI-powered system delivering continuous improvement after every execution, all that’s needed is for the system to review the app once and extract all properties related to the element. Once this is done, there is no need to author tests using the visual aspects of the app. Technological advancements indicate that this would become feasible in the not too distant future.



Semi-Autonomous (Level 2)

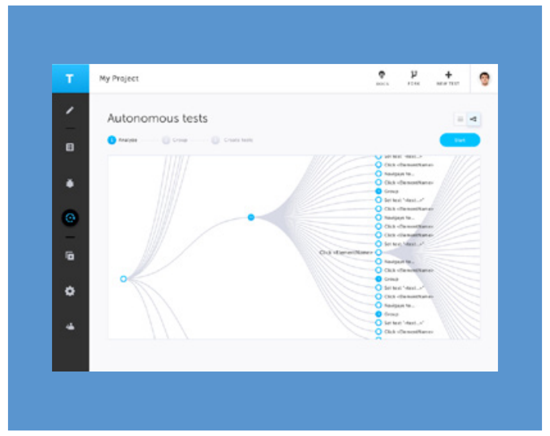
In this instance, tests can be authored by the system when it connects to staging or production environments and observes the way humans interact with the app — either QA/dev team interacting with the app in one of the development cycles or real users during production. One of the major improvements of Level 2 over Level 1 is that AI understands that there are repeatable actions. So, it can help cluster these actions into groups and convert them into reusable user scenarios.

Fully-Autonomous (Level 3)

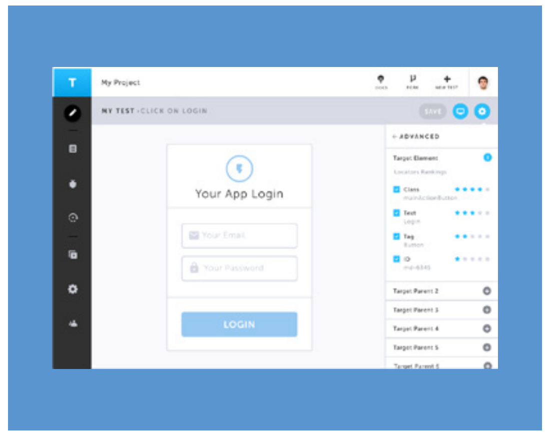
At this level, there is no need for human intervention — the system undertakes test authoring by itself (except where login credentials are needed). This level could also be described as monkey testing where the app clicks randomly to go through the various states of the application. While this can be relatively easy for simple apps, it can be extremely difficult for others...for instance, a complex application like Paint.

While there are no major differences between validation and a wait-for (inability to reach a specified state means failure in both instances), we are going to separate them into two sections.

LEARN BY OBSERVATION & REAL CUSTOMERS ON PRODUCTION



AGGREGATE USER ACTIONS INTO FLOWS (REUSED COMPONENTS)



TESTS PRODUCED FROM (REUSED) FLOWS

Wait fors

Since virtually all apps are synchronous, users are only able to click on elements by following visual prompts on a screen. Most platforms make use of implicit waits — where no action is taken until the element becomes visible — to achieve this. In most cases, this means waiting for a specific state (such as waiting for a dropdown to populate with a list that was dynamically requested from the application's database/backend).

Manual Authoring

At this point, tests are authored manually and no attention is paid to wait-fors until tests begin to fail. Most testers add "sleep for X seconds" functions in their apps resulting in flakey tests. Also, the addition of such random sleeps can negatively impact the speedy execution of tests.

Teach by example

An AI-powered system that observes live users interacting with applications can see patterns and derive insights much better than humans. Such patterns could include instances where database/ network requests have been sent and need to be processed or wait-fors where elements must become visible before subsequent actions can be performed.

Semi-Autonomous

The system could review previous failed iterations and suggest improvements to test authoring.

Fully-Autonomous

Since the system is able to randomly add delays between executions and train automatically, it becomes fairly easy to achieve a fully autonomous system.

Validations

After performing the necessary actions, the next step involves validation. This means checking to see if the intended state has been achieved. This involves three steps:

- Conducting text validation at the DOM level (regex, string compare, etc.).
- Validating CSS properties(distance between elements, color, height, etc.).
- Checking the rendered pixels.

Element style and text validation involves checking to see if certain elements in the UI have a specified value while pixel validation involves taking a snapshot of all or part of the screen and comparing it to a baseline that contains the expected results.

Style validation

The state of all visual properties is visually tested by style validation. One example of this kind of validation is the Galen framework where you can verify certain properties — for instance, verifying that the width of an element is 15px or is located a certain distance from a neighboring element.

Manual Authoring

A lot of software development organizations are still at this level where validation requires a substantial amount of manual effort — usually accomplished by extracting text from the UI and using a relevant function (such as an ad hoc regex or the equal method) to compare values.

However, Pixel Validation and Style Validation are fragile and cumbersome methods and not recommended by industry experts. Style validation creates numerous validations for every page, making it cumbersome to maintain while the issue with pixel validation arises from two major factors

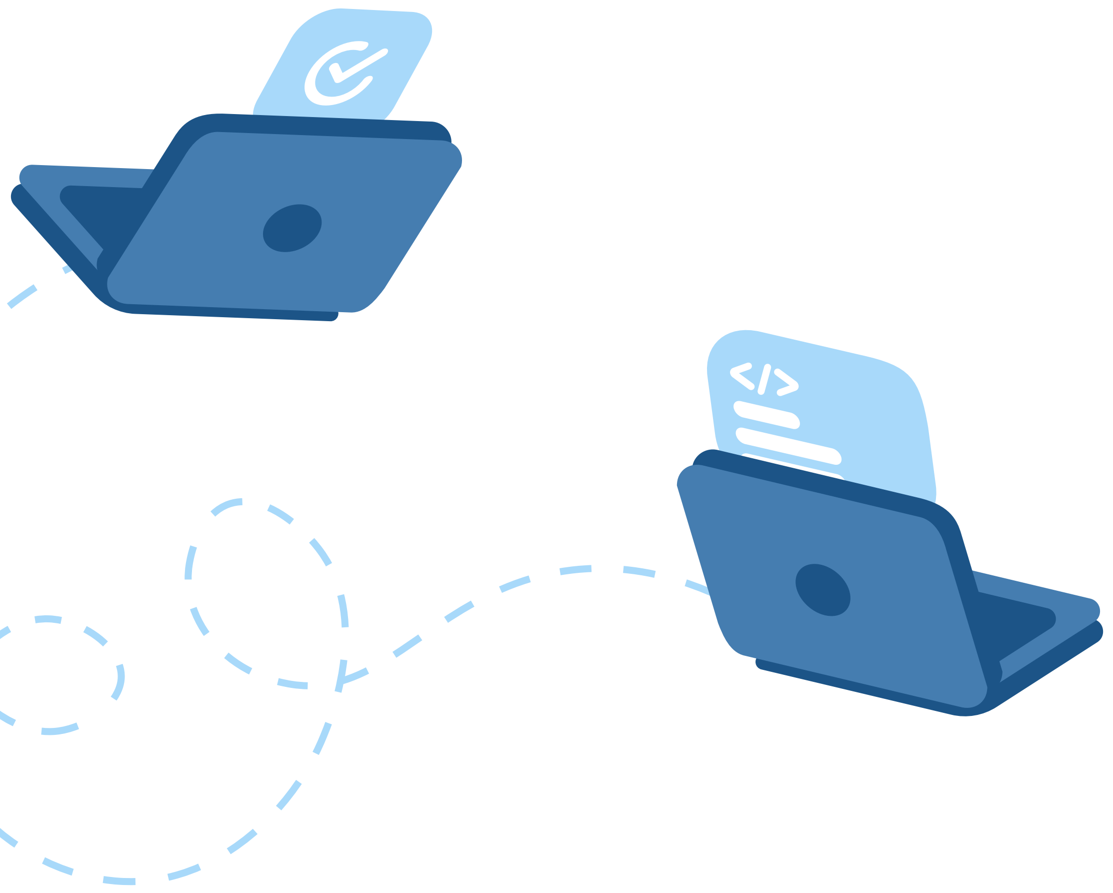
- The non-deterministic nature of display adapters which generates slightly different results for each iteration. However, these variations (caused by subpixels shifts and anti-aliasing) are unnoticeable to the human eye.
- Too much validation caused by taking a snapshot of the entire page (every location, color and text). This leads to high maintenance especially for pages where elements change frequently.

Teach by example

At this point, human intervention is still needed to help filter out regions consisting of random values (like time and date) and other types of noise. Pixel validation still remains the focus at this stage where improvements in the system's prediction abilities results in much lower false positives.

TDD

As the system's accuracy in anticipating variations continue to increase, it will become easier for dev teams to specify the level of variations that acceptance tests can accommodate. Style validation can be invaluable at this point where design editing software (e.g. Photoshop) can be used to directly generate and add it to the test.



Semi-Autonomous

By enabling automatic maintenance for validations, we're gradually approaching the semi-autonomic point. For instance, changing logos can affect up to 5000 UI screens. However, the system can determine that such a modification is a single universal change, rather than a bug. As such, there's no need to review 5000 snapshots and manually approve each one.

Fully-Autonomous

In this instance, deep learning can get us closer to where we want to be — it can automatically verify that a page's alignment, font size, and structure looks good or help detect any problems. However, it would be challenging (nigh impossible) to automatically generate accurate answers to every query.

Setup

This relates to an application's starting point before the execution of action(s) that modify its initial state.

Manual Authoring

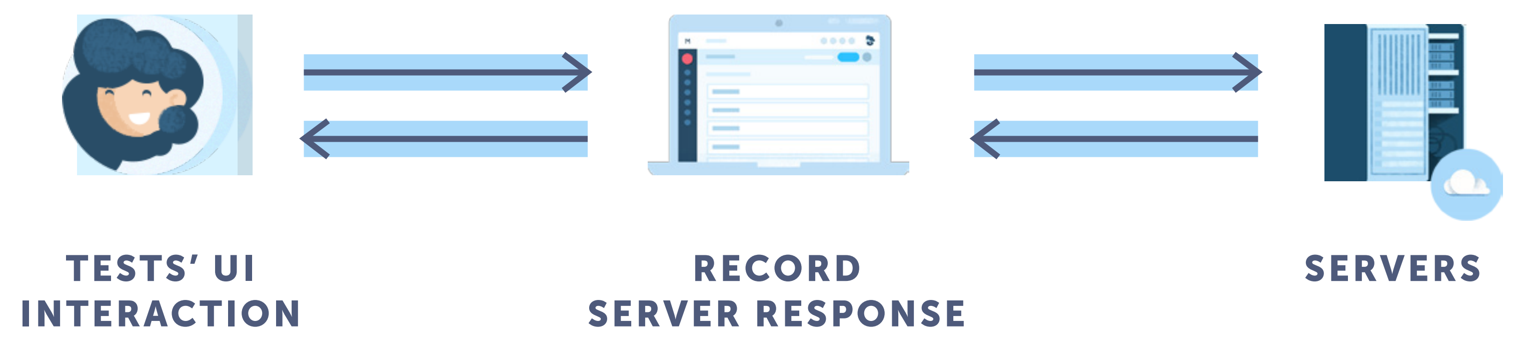
Most inexperienced testers fail to take into consideration an application's initial state when authoring tests, thus leading to a lot of failed tests. This is problematic for most organizations, especially those that write their own ad hoc code.

Teach by example

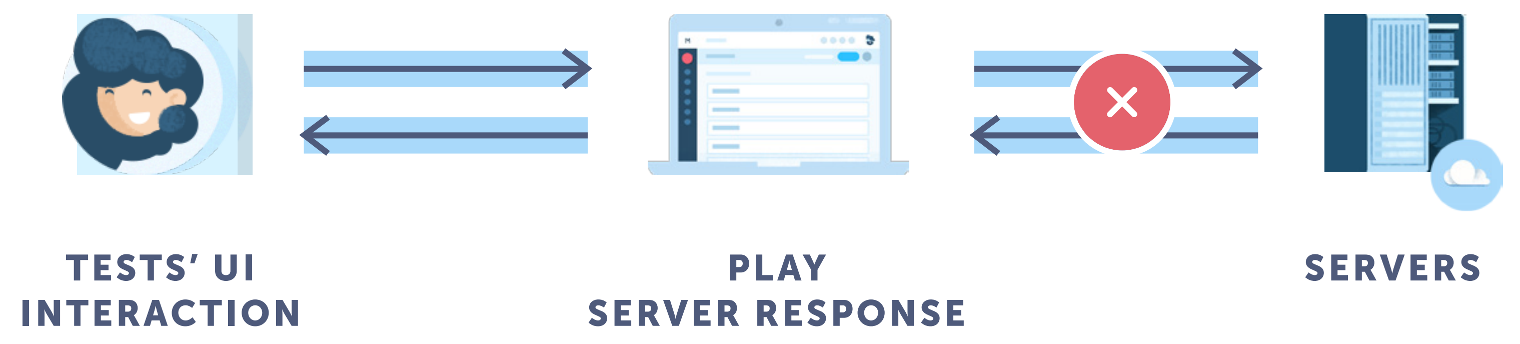
While dumping and restoring DBs before each test run appears easy for most systems, failures will start occurring when tests are run in parallel. The solution is leveraging optimization to review the database and determine the minimal projection needed to execute tests. However, this isn't feasible in load testing applications and could result in more problems due to the precipitation of false negatives. As such, this option should only be used to speed up test optimization in the early stages, not for full acceptance tests.

AUTOMATIC RESPONSE

1 CALL TO SERVER ARE RECORDED



2 RESPOND WITH MOCKS



Semi-Autonomous

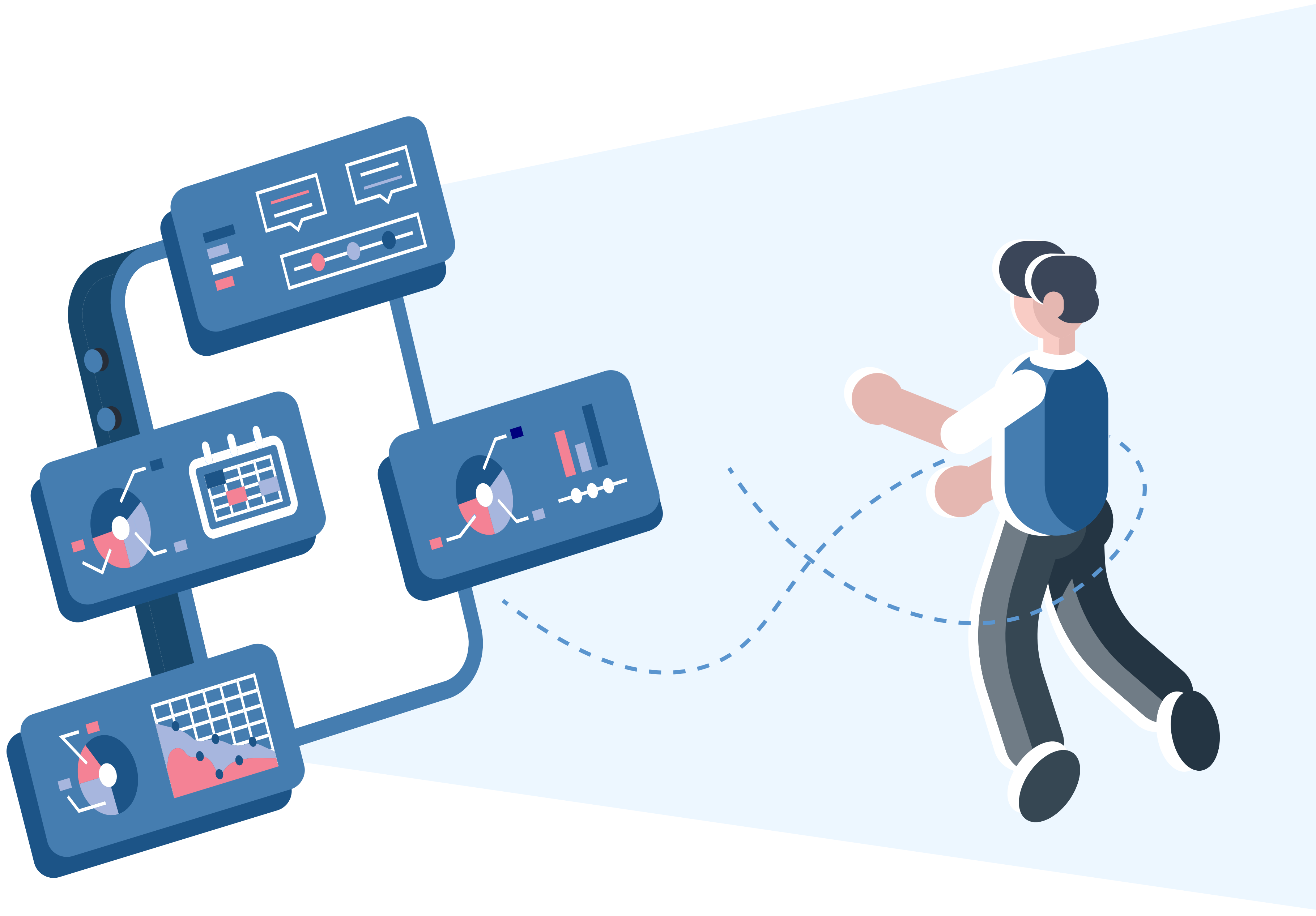
The system can create mockups to enable faster testing by observing the network and saving ongoing communication between the app's components and services. For instance, if there were code changes on the app's front end, it's faster and more efficient to have calls to the server or microservices mocked up and replaced with saved data than setting up a database and server. This is particularly useful in cases where we want to recreate an instance where the server returns an error and the validated error message is displayed to the user. While this will require some level of human intervention, AI can help generate dynamic content that users will understand and automatically alter the response depending on the kind of error encountered.

Fully-Autonomous

One of the best ways to generate test cases (and the data needed for test executions) is by using model-based testing. Although model-based testing is mature, it is a rarely used method because dev teams must model the entire app being tested. This fact hinders its adoption and greatly reduces the ROI.

By leveraging AI, systems can understand the interrelationship between components, services and objects thereby automatically creating the model and making it easier to automate test authoring.

While this sounds complex and implausible, there are indications that semi-autonomous versions (which observes user interactions with apps and derives accurate insights and predictions) will be released very soon.

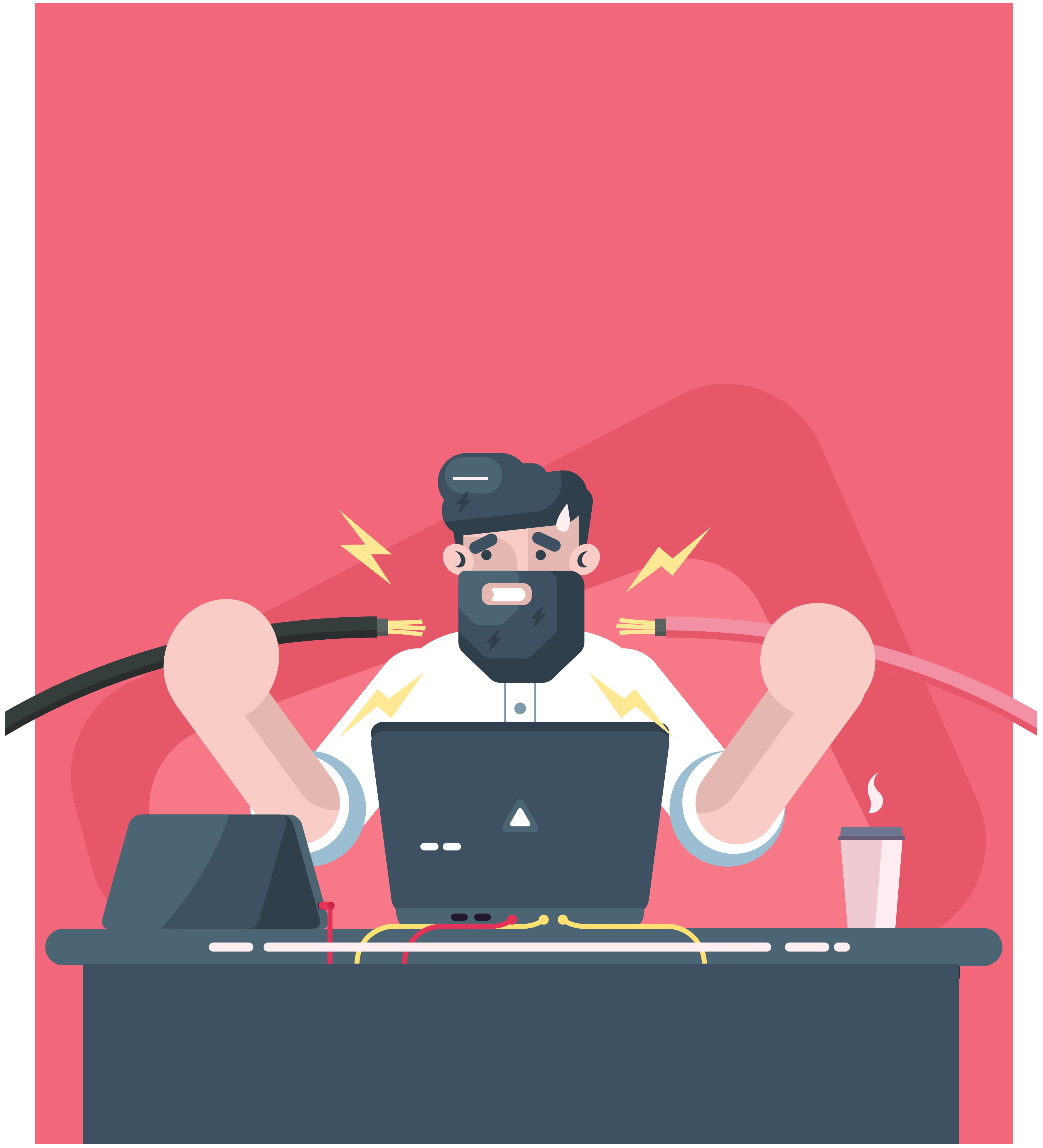


Risk Management

QA is all about the needs of customers and end users. Whether it is testing software functionality, enabling a flawless user experience or protecting their data, it all comes down to managing risk. While many companies leverage code coverage as a way to reduce risk in functional testing, there's a much better approach.

Connecting your apps and production environment to your testing cycle will not only help resolve challenges with test authoring but they can give dev teams insights into areas where they should focus their testing on (for regression).

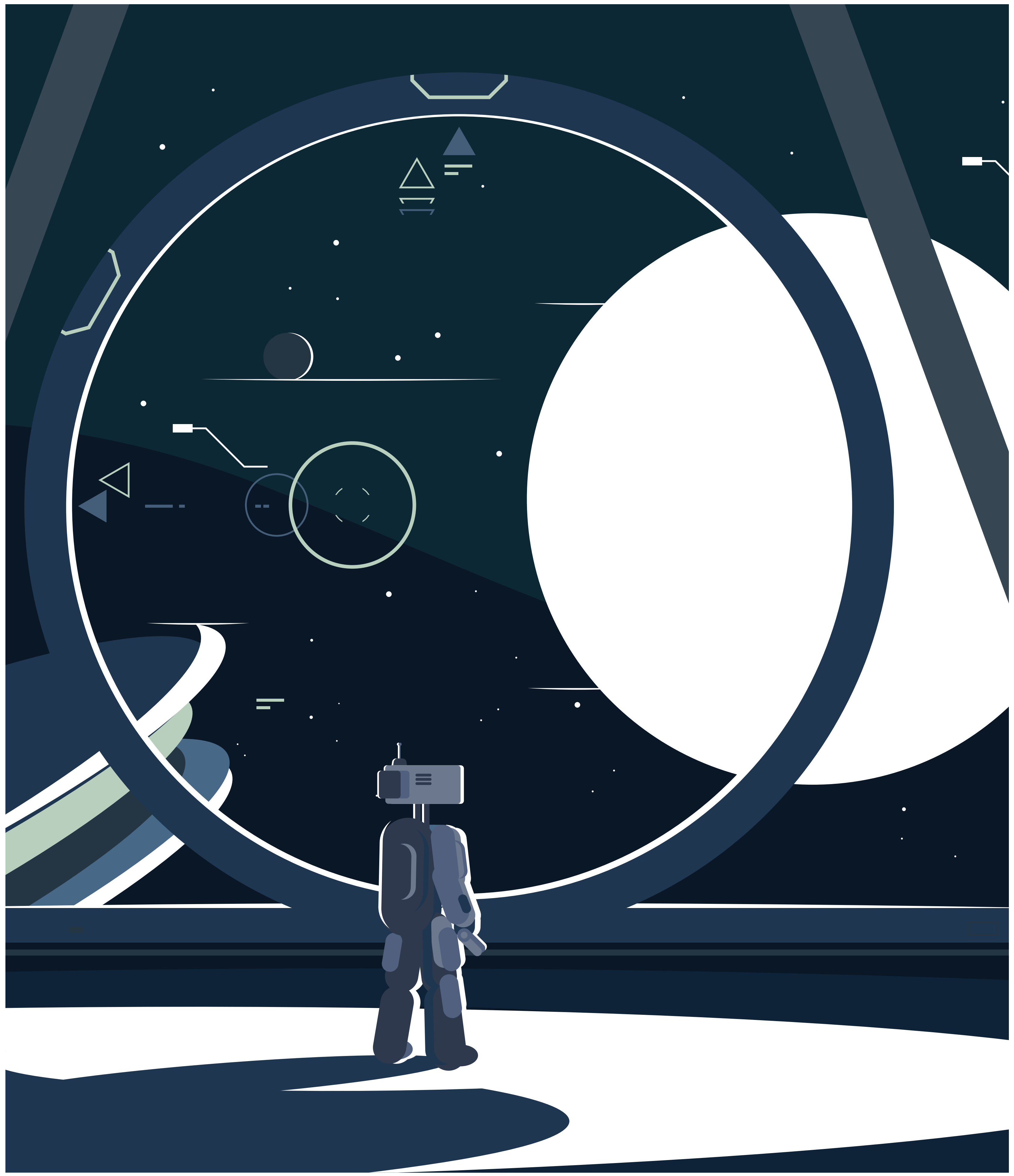
To maximize productivity, it's essential that teams review the areas that are most critical to the success of a business or at least, those scenarios that occur most frequently. Hopefully, there will be less focus on code coverage and more on user coverage in order to test what customers are going through.



Summary

With AI helping dev teams gradually achieve Autonomous Testing, it is hoped that the quality/velocity dilemma facing software development organizations will finally be eliminated. Autonomous testing will improve software quality by helping to connect production apps and test authoring before mapping them to real user flows, making it easier to maximize user coverage. Also, it will facilitate a risk-based approach thus enabling engineering teams make better data-driven decisions.

At Testim, our biggest differentiator is the use of AI to proactively fix issues (through a self-healing mechanism) while reducing the amount of maintenance to be done by our clients. We also increase development velocity for our customers by making it easier to author user scenarios within shorter periods of time, thus increasing release velocity and enabling faster time to market. In essence, our autonomous testing tool helps facilitate a much brighter future for software quality.



About the Author

Oren Rubin

Oren has over 20 years of experience in the software industry, building mostly test-related products for developers at IBM, Wix, Cadence, Applitools, and Testim.io. In addition to being a busy entrepreneur, Oren is a community activist and and the co-organizer of the Selenium-Israel meetup and the Israeli Google Developer Group meetup. He has taught at Technion University, and mentored at the Google Launchpad Accelerator.

testim

Thank You

For more information
contact us at info@testim.io