O'REILLY®

Compliments of KEYSIGHT TECHNOLOGIES

AI-Driven Testing

Bridging the Software Automation Gap

Tariq King

REPORT

The new Eggplant Salesforce Solution auto-generated test assets to accelerate your Salesforce application releases

KEYSIGHT TECHNOLOGIES

Assess Your Salesforce Instance Today | eggplantsoftware.com/salesforce-solution

AI-Driven Testing

Bridging the Software Automation Gap

Tariq King

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

AI-Driven Testing
by Tariq King

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Rebecca Novack
Development Editor: Virginia Wilson
Production Editor: Christopher Faucher
Copyeditor: nSight, Inc.
Proofreader: Paula L. Fleming
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Kate Dullea

June 2021: First Edition

Revision History for the First Edition
2021-06-22: First Release

978-1-098-10597-6

[LSI]

Table of Contents

Preface

Software testing is expensive. Testing requires both up-front and recurring investment in labor and assets to reduce the risk of shipping a product that does not meet customer expectations. Good testers and engineers with knowledge and experience applying software-testing best practices are hard to find. By the end of a project, you've spent 30%–50% of your engineering budget on software-testing activities,[1] including requirements and design testing, unit testing, user acceptance testing, performance testing, and security testing.

Not investing enough in testing can have an even worse impact on your organization's bottom line. The cost of fixing defects increases exponentially with the time and stage at which they occur during the development cycle. Bugs found early during requirements elicitation are much cheaper to fix than those discovered while coding or, worse yet, after the software is released. Bugs that do escape to production directly impact customer satisfaction and can ultimately cost you your reputation and business.

The key to controlling quality cost-effectively is to find the right level of testing effort for your project based on the risks associated with the release. Ideally, you want to do just enough testing that you can remove the most harmful defects prior to the release. Of course, the more test coverage, the better, but if that coverage comes with too great an investment cost, you end up overtesting and not getting a good return on your investment (ROI). In other words, over time,

1 Capgemini, *World Quality Report 2019–20*, 11th edition, 2020.

it becomes harder and harder to find that next defect, and eventually the cost to find and fix the problem may not be worth it.

One way to reduce testing costs and improve your ROI is through automation. By removing manual steps, organizations can scale operations faster, easier, and more cost-effectively. Unfortunately, the current state of the art in test automation still requires significant manual effort. Humans have to first understand the software requirements, design and specify test cases, and then manually translate them into machine-readable scripts. A software-testing tool or framework then executes the scripts against the system under test and logs the results. However, the moment the script ends, humans enter back into the loop to verify the results and translate them into actionable items. The good news is that advances in artificial intelligence (AI) and machine learning (ML) are being used to bridge the gap between manual and automated software testing.

Who This Report Is For

If you are a technology leader, such as a CTO, VP of engineering, quality director, or engineering manager responsible for developing high-quality software, who is interested in learning how to scale those efforts cost-effectively through test automation, then this report is for you. Other roles that may benefit from this report include test automation engineers, software engineers, architects, technical leads, or researchers who are using or developing automated software-testing solutions.

What You Will Learn

In this report, I will discuss the grand challenges and limitations of traditional automated testing tools and describe how AI-driven approaches are helping to overcome these problems. We will explore the application of AI/ML to functional, structural, performance, and user-design testing, and we'll dive into techniques for automating graphically intensive solutions such as video streaming and gaming applications. By the end of the report, you will have a wide knowledge of the various applications of AI-driven testing, an understanding of its current benefits and limitations, and insights into the future of this emerging discipline.

CHAPTER 1

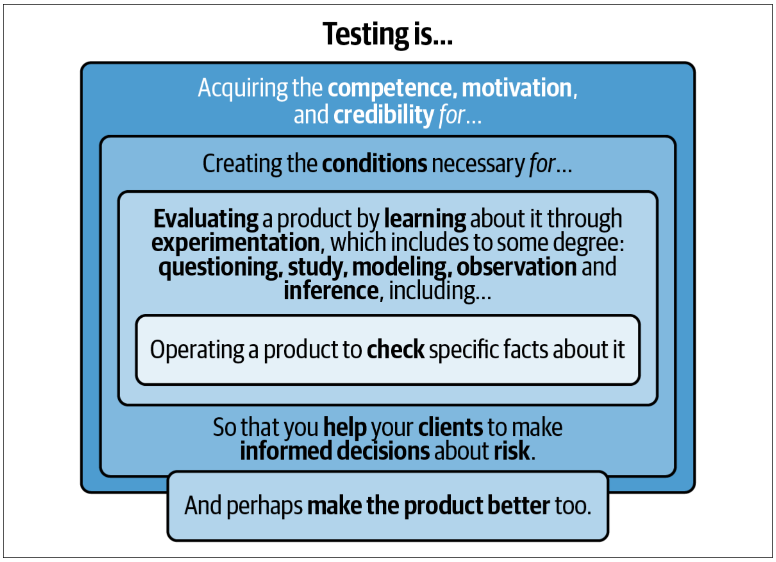
The Test Automation Gap

When you automate something, it can start, run, and complete with little to no human intervention. In the 1950s, the term "lights-out manufacturing" was coined to describe a vision of factories so independent of human labor that they could operate with the lights out. However, in the field of software testing, our view of automation is far from a lights-out philosophy. In this chapter, I describe the gap between manual and automated testing, including some of the grand challenges of software testing and limitations of traditional test automation approaches.

The Human Element of Software Testing

For years, you've likely differentiated between manual and automated testing. In my own career as a test architect, engineering director, and head of quality, I frequently differentiated between the two, even becoming good at convincing others that making such a distinction was both meaningful and necessary. However, the truth is that, outside of automatic test execution, software testing is almost entirely a manual, tedious, and time-consuming process.

Before you can test a software system, you must understand what the system is supposed to do and who its intended users are. This generally involves reviewing software requirements documentation; interviewing product analysts, end users, and customers; and comparing this information to the requirements of similar products. Once you understand the product requirements and customer needs, you must examine the implemented system to determine whether or not it meets those needs. Software testing is more than having a human and/or machine check specific facts about a program.[1] Figure 1-1 visualizes a definition of software testing, reminding us that software testing is a complex, multidimensional problem that is as much social as it is technical.



Testing is...
Acquiring the competence, motivation, and credibility for...
Creating the conditions necessary for...
Evaluating a product by learning about it through experimentation, which includes to some degree: questioning, study, modeling, observation and inference, including...
Operating a product to check specific facts about it
So that you help your clients to make informed decisions about risk.
And perhaps make the product better too.

Figure 1-1. Testing is more than checking—it involves learning, experimenting, and more

Grand Challenges

To truly automate testing, we first have to overcome some of its key challenges. Let's take a look at five grand challenges in software testing that make it difficult to automate.

Input/Output

Computer software takes user input, processes it, and produces output. Let's assume you are testing a very simple program with only one text field. For that single field, you have a computationally infeasible number of values to choose from. Considering that most programs contain many input fields, now there is a combinatorial

1 James Bach and Michael Bolton, Rapid Software Testing, accessed June 3, 2021.

explosion of the computationally infeasible number of input values. Furthermore, you can apply those input combinations at different times and order them in many different ways. So now there is a permutational explosion that must also be addressed. Sounds exhausting, doesn't it? And don't forget that you still have to check that each of those inputs produces the correct output and does so in accordance with the appropriate usability, accessibility, performance, reliability, and security constraints.

State Explosion

Most computing systems are reactive. They engage users in a session where a human or another system provides input, the machine processes it and produces output, and this process then repeats. At any point, the system is in a state where, depending on previous inputs, a given user action will produce specific outputs and the system will transition to an updated state. If you are testing a reactive system, you are responsible for getting the system into the correct pretest state, applying the input, and verifying that the correct end state has been reached. However, an infinite number of inputs and outputs is generally accompanied by an infinite number of states. And so, similar to the input/output problem, testing suffers from a rapid explosion in the number of states you have to cover.

Data

During testing, you can use data in many different ways. You may want to confirm specific facts about how the system transforms inputs into outputs or set up unusual scenarios to verify that the system does not crash. Testers use data selection techniques like equivalence class partitioning[2] to reduce the number of tests required to cover a particular feature and choose data around the system boundaries to verify that the software works under extreme conditions. Yes, data is a powerful tool in your testing tool belt. However, this also means that there are many important decisions that surround that test data. Will your test data primarily consist of fake data that may not be representative of the real world? Or are you

2 *Equivalence class partitioning* divides the input space of a program into different sets, each containing behaviorally equivalent inputs. The idea is that you can reduce the number of tests needed to cover a program by testing one value from each partition instead of trying to test each and every input.

able to pull real-world data from customers in production? If so, can you anonymize customer data for testing in accordance with data privacy regulations? The point is that, even though it is possible to build tools to automatically generate data, there are so many factors to be considered that humans generally need to be kept in the loop.

The Oracle

If you've watched the popular movie franchise *The Matrix*, you know that the Oracle was a character with highly accurate insights and predictions. She seemed to know how the system would or should work, and everyone went to her for the answers. Sometimes her responses felt vague and weren't exactly what people expected to hear. The similarities between this fictitious movie character and the oracle problem in software testing is uncanny. When we refer to the *oracle* in testing, we are talking about knowing whether or not the system did what it was supposed to do. Human testers do their best to act as an oracle but generally rely on other domain experts. Answers about the system's expected behavior tend to be vague and subjective, and, as a result, creating an automated oracle is a non-trivial problem.

Environment

Software does not live in isolation but is deployed in an environment. One of the value propositions of testing over other validation and verification techniques is that testing provides information about the software running in an actual environment. The challenge is that the environment itself is a combination of hardware and other software and it varies according to the client and usage scenarios. The hardware environment includes physical devices such as corporate servers, workstations, laptops, tablets, and phones, while operating systems, virtual machine managers, and device drivers make up the software environment. Obviously, it is impractical to test all of the hardware and software combinations, so the best practices tend to focus on risk and past experiences.

Limitations of Traditional Approaches

Traditional approaches to software test automation are hyperfocused on developing machine-readable scripts that check for specific system behaviors. The irony of the situation is that creating these so-called automated scripts requires significant manual effort. Recall that testing involves learning by experimentation, study, observation, inference, and more. Without automating these highly cognitive functions, it is only possible to develop test scripts after a lot of the hard work to test the product has already been done.

You first have to understand the requirements and implemented system and then devise test cases that include preconditions, test steps, input data, and the expected oracles. Once you've reviewed and iterated your test cases, you can manually encode them as scripts. The good news is that with the scripts in hand, you now have the ability to rerun these specific test scenarios on subsequent software releases. The bad news is that traditional tools and frameworks are quite rigid, so there are high maintenance costs associated with updating and maintaining these scripts over time. Note also that these scripts generally run the same scenarios using the same data each time. As such, most traditional approaches suffer from a phenomenon known as the *pesticide paradox*. The paradox states that, just as insects build up resistance to pesticides over time, the more you run the exact same tests on your software, the more likely it is to become immune to issues surrounding those specific scenarios.

When it comes to automation, the attention of the software-engineering and testing communities has primarily been on functional user interfaces (UIs), service, and code-based testing, and nonfunctional concerns such as system performance and security. To close out this chapter on the test automation gap, let's look at the limitations of the current state of the art in test automation with respect to these three dimensions.

UI Test Automation

Functional UI testing seeks to determine if the software is able to carry out its required functions from the users' perspective. Functional UI automation mimics users clicking, typing, tapping, or swiping on various screens of the application and verifying that the appropriate responses appear. For web and mobile applications, testing frameworks such as Selenium and Appium locate screen elements using patterns and path expressions based on the application's document object model (DOM). One of the major drawbacks of DOM-based element selectors is that they make tests susceptible to breaking as the structure and/or behavior of the UI changes. For example, changing the location or identifiers for existing application screen elements and navigation paths generally means that you have to update all the page object models or test scripts associated with them. Furthermore, since most of the functional UI automation tools out there target applications with a DOM, there are few to no end-to-end automation solutions for graphically intensive applications.

Service and Unit Test Automation

Of course, not all testing is done via a graphical user interface. In fact, best practices like the test pyramid[3] promote the idea that the majority of your test automation should be done at the API—commonly referred to as service—and unit levels. Although some tools make use of schemas and constraint solvers for API- and code-based test generation, the primary focus at this level continues to be on writing scripts that execute predefined tests and measuring code coverage. However, code coverage can be a misleading quality measure if you do not use it correctly.[4] Instead, practitioners recommend that you aim for better coverage by testing the software on a variety of input values, which should lead to improvements in code coverage. In other words, let coverage of the input space, rather than coverage of the code, be a key goal in your testing strategy.

Nonfunctional Test Automation

Nonfunctional testing validates constraints on the system's functionality such as performance, security, usability, and accessibility. End-to-end performance testing tools capture network traffic and facilitate the development of automated scripts that simulate user actions to check performance, scalability, and reliability. The idea is to emulate production traffic using hundreds and thousands of virtual concurrent users performing real-world transactions. Once the system is under load or stress, you can monitor performance metrics such as response times, memory usage, and throughput across servers and other network resources. Just like functional UI test scripts, performance test scripts require periodic maintenance but

3 Mike Cohn, "The Forgotten Layer of the Test Automation Pyramid," Mountain Goat, December 17, 2009.

4 Brian Marick, "How to Misuse Code Coverage," Exampler Consulting, Reliable Software Technologies, 1999.

also introduce an additional layer of complexity. For example, if you add new application screens or navigation paths, not only do you have to update the script's element selectors and test steps, but you must also recapture the network traffic to reflect the updated transactions. Traditional automation approaches to security testing are usually limited to injection attacks, code scanners, and fuzzing tools. Areas such as usability and accessibility are often deemed too difficult to automate because of their qualitative nature. Designs are not necessarily correct or incorrect but rather good or bad, and how good or how bad may depend on whose opinion matters most.

Conclusion

Automation in any field has limitations and may require manual effort to set up or guide the process. However, my experience with software test automation in the field over the last 15 years has led me to believe that many practitioners have accepted a subpar definition of what it means to automate the testing process. On the other hand, other practitioners claim that testing is so hard that it will never be automated. Regardless of which school of thought is right or wrong, understanding the test automation challenges and limitations described in this chapter is the first step on the road to narrowing any gaps in the current state of the art and practice of software testing.

CHAPTER 2

Leveraging AI for Test Automation

Did you know that autonomous and intelligent agents, commonly referred to as bots, are already running tests on major applications today? That's right: leveraging AI for software testing is not a thing of the future; AI for test automation is already here. The bots are in the building, and they're not testing just one app but many apps in various application domains.[1] In fact, don't be surprised if you find out that AI is already testing your own app! Chances are that, if you publish your app in one of the major app stores, AI bots are already testing it. In this chapter, I'll walk you through how AI tests software and demystify how this technology really works when testing applications at different levels or for various quality attributes.

AI for UI Testing

Just because certain tasks have historically required human effort does not mean that we won't be able to automate them someday. Once upon a time, we believed that tasks such as voice and image recognition, driving, and musical composition were too difficult for computers to simulate. However, many of these tasks are now being automated using the power of AI and machine learning. In some cases, AI is outperforming human experts in tasks such as medical diagnosis, legal document analysis, and aerial combat tactics, among others. With that in mind, it really shouldn't surprise you that we're

1 Test.ai, "Case Study App Store Provider," 2020.

leveraging AI for functional testing tasks that previously relied on the expertise of human testers. Figure 2-1 illustrates how to train AI bots to perceive, explore, model, test, and learn software functionality. It is important to note that even though learning through feedback is explicitly called out at the end, the bots leverage machine learning at each stage of the process.

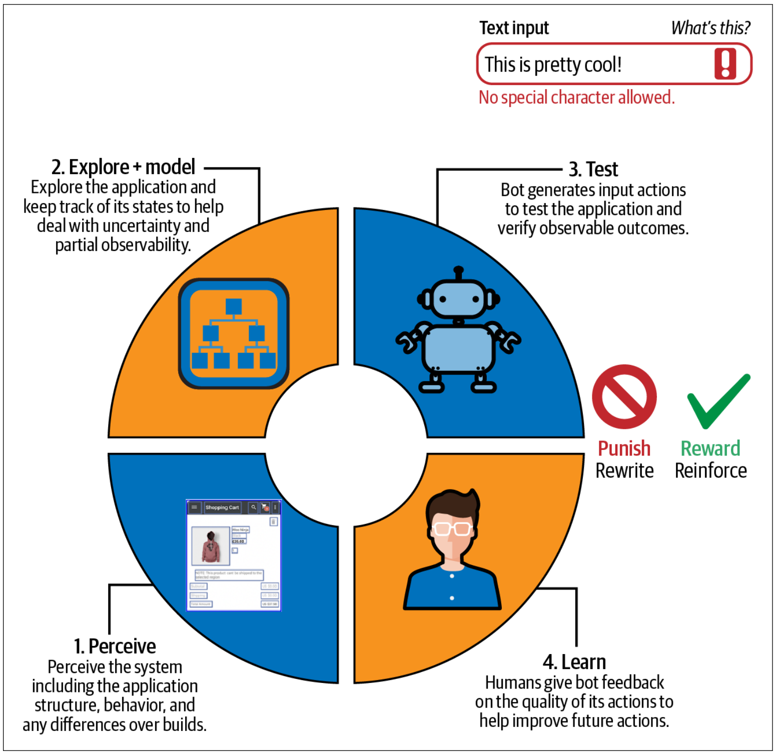

Text input
What's this?
This is pretty cool!
No special character allowed.
2. Explore + model
Explore the application and keep track of its states to help deal with uncertainty and partial observability.
3. Test
Bot generates input actions to test the application and verify observable outcomes.
Punish
Rewrite
Reward
Reinforce
1. Perceive
Perceive the system including the application structure, behavior, and any differences over builds.
4. Learn
Humans give bot feedback on the quality of its actions to help improve future actions.
Shopping Cart

Figure 2-1. Training AI to do functional UI testing

Perceive

A foundational step in functional UI test automation is having the ability to interact with the application's screens, controls, labels, and other widgets. Recall that traditional automation frameworks use the application's DOM for locating UI elements and that these DOM-based location strategies are highly sensitive to implementation changes. Leveraging AI for identifying UI elements can help to overcome these drawbacks. Just like humans, AI bots recognize what appears on an application screen independently of how it is implemented. In fact, there need not be a DOM at all, as the interaction can be based solely on image recognition. AI, and more specifically a branch of AI known as *computer vision*, gives us the test automation superpower of being able to perceive anything with a screen. Furthermore, since we train the bots on hundreds and thousands of images, UI design changes do not result in excessive test script maintenance. In many cases, AI-based test automation requires zero maintenance after visual updates and redesigns.

Explore and Model

Testers frequently explore the application's functionality to discover its behavior, confirm specific facts, and look for bugs. While exploring, they create mental models and refer back to these models to deal with uncertainty when the application or its environment changes. Similarly, AI bots explore and build models of the application under test. You give them a goal and they attempt to reach it by trial and error, a process known as reinforcement learning. An easy way to understand how reinforcement learning works is to think about how you train a pet dog. To start, you decide on the task you want the dog to accomplish—let's say "stay." You also need a bag of treats. If by chance you say "stay" and the dog sits in place, you give it a treat. However, if the dog continues to move around, you keep the treat. You could even take it a step further by emphasizing "bad dog" and showing your discontent.

Figure 2-2 provides an illustrative example of how to leverage goal-based reinforcement learning for exploring, modeling, and testing a software application. To start, all the bot needs is the location of the application's home screen and for us to give it an objective. Let's task the bot with navigating to the shopping cart. In our initial state (a), the bot is on the HOME screen and has the goal of reaching the checkered flag on the CART screen. Don't let this visualization deceive you; for now, the bot only knows about the HOME screen and as a result has only this single state in its application model. Recall from the previous subsection that it can recognize the screen and its various widgets. Another prerequisite I introduce here is that the bots must be able to stimulate via input actions such as keystrokes, taps, clicks, and swipes. Prior to its taking any input actions, I give the bot an initial score of zero. Scoring represents the reward system, a bag of treats, so to speak, which can be positive or negative. Now the bot takes its first action. It randomly clicks a link and transitions to the PRODUCT screen (b). The bot has now seen two screens, HOME and PRODUCT, and updates its application model with this information. My response to the bot's actions is that this isn't really what I want—I am really looking for the CART—so I deduct 1 from the bot's score. From PRODUCT, the bot takes another random action, and this time it lands on the CART (c). Excellent! This is exactly where I want the bot to be, so I reward the bot with 100 points. Following this path, the bot ends up with a score of 0 – 1 + 100 = 99 points and a complete model of the application. Let's call this exploration scenario Episode 1.

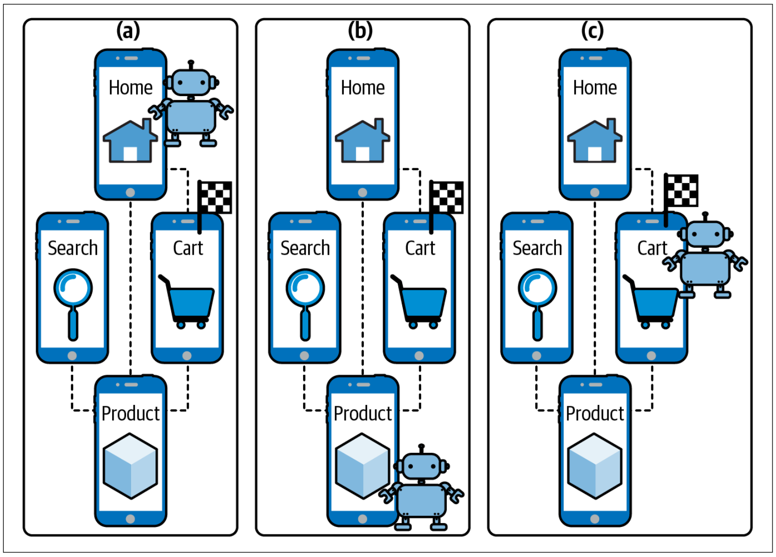


Figure 2-2. Illustrative example of using bots to explore an app using reinforcement learning

Consider a second exploration scenario, where the bot once again starts on the HOME screen (a). However, instead of navigating to PRODUCT, the bot takes an action that takes it directly to the CART (c). Applying the scoring system, the bot earns 0 + 100 = 100 points for Episode 2. The bot essentially finds a path with a higher reward and, moving forward, will follow that path to accomplish its task. In short, the bot learns by combining trial and error with past experiences rather than taking a brute force approach, which as you may recall is computationally infeasible. Goal-based reinforcement learning is practically applicable to a variety of testing tasks, making it an extremely powerful technique for test automation.

Test

Now that the bots can perceive, explore, and model the application, these capabilities come together for the greater good of software testing. The bots are trained how to generate specific types of input data and can recognize what expected and unexpected behaviors look like in given contexts. Note that it may be easier for bots to identify some types of issues than others. For example, an HTTP 404 error is an obvious indication that the application has thrown an error. However, it is significantly harder to know that someone's pay stub is incorrect because their taxes weren't calculated appropriately. Nonetheless, several researchers and practitioners are applying AI/ML research to automatic functional UI test generation. This work ranges from generating inputs for individual fields to conducting complete end-to-end test cases, including oracles.[2] Although we have only scratched the surface in this area, AI-based test generation approaches are slowly narrowing the test automation gap.

Learn

One of the most notable characteristics of AI- and ML-driven applications is the ability of the system to improve how it performs a given task based on feedback. Having humans provide direct feedback on the bots' actions—for example, recognizing UI elements, generating inputs, or detecting bugs—makes the system better. Feedback mechanisms allow humans to reinforce or rewrite the AI brain that drives the bots' behavior. As a result, the more feedback your teams provide to the bots on the quality of their testing, the better the bots become at testing the product, and ultimately the more value they provide to the testing team. Typically, feedback is incorporated into the product UI itself. However, depending on the level of testing, feedback may come via mechanisms for updating datasets more directly.

2 Dionny Santiago, "A Model-Based AI-Driven Test Generation System" (master's thesis, Florida International University, September 9, 2018).

AI for Service/API Testing

Automated service or API testing validates the way systems communicate via sequences of requests and responses. For example, a typical communication exchange between two services, A and B, could be as follows:

1. Service A sends an HTTP GET request to retrieve some data from Service B.
2. Service B processes the request and returns an HTTP status of 200, indicating success along with a response body containing the requested data.

A common approach to automated API testing with AI is to record the service traffic from manual testing and then use that information to train an ML-based test generation system. One of the key goals for this type of testing is to verify that the communication sequences do not result in unhandled errors. This approach stems from the once popular record-and-playback feature found in early functional test automation tools, so you'll see this pattern in other areas such as performance testing. However, although there are several available open source and commercial API test automation tools on the market,[3] few of them offer these new "record, learn, generate, and play" capabilities. Since automated tests, like those for APIs, map naturally to interleaving sequences of inputs and expected outputs, another AI-based approach that applies to API testing involves using *long short-term memory machines* (LSTMs) for generating test sequences.[4] Under this approach, you would train the machine on string sequences of abstract test cases that contain example service requests and their respective responses. Figure 2-3 depicts the workflow for developing and validating such a test flow generator using neural networks.

3 Joe Colantonio, "Top API Testing Tools for 2020," Test Guild, May 16, 2017.

4 Dionny Santiago, "A Model-Based AI-Driven Test Generation System."

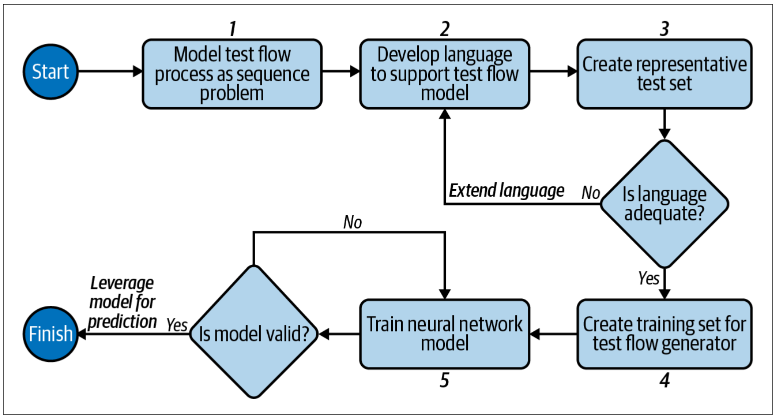

Start
1
Model test flow process as sequence problem
2
Develop language to support test flow model
3
Create representative test set
Is language adequate?
No
Extend language
Yes
Create training set for test flow generator
4
Train neural network model
5
Is model valid?
No
Yes
Leverage model for prediction
Finish

Figure 2-3. Automatic test case generation using neural network models

These are the major steps of the workflow in the context of API testing:

1. Model the API testing flow as a sequence problem.
2. Develop an abstract test language to support the API test flow model.
3. Create a test set to validate the adequacy of the language in describing API tests.
4. Curate and/or gather example handcrafted API test flows.
5. Train a neural network to generate valid test flow sentences that belong to the grammar of the abstract test language.

With an abstract test case generator for your APIs in place, you can now develop an engine that transforms those abstract, platform-independent tests into platform-specific tests for execution using any given communication protocol or technology.

AI for Unit Testing

Researchers and practitioners are training AI to automatically write unit tests in high-level programming languages like Java.[5], [6] Much of the work in this area builds on advances in AI for generating text and natural language. Initiatives like Open AI's GPT-3 combine natural language processing (NLP) and deep learning models to generate text that is difficult to distinguish from human-written text.[7] Here are two popular features of AI for unit-testing tools and frameworks:

- Automatic source code analysis to generate unit tests that reflect program behavior and help to reduce gaps in coverage.
- Integration with version control systems to monitor source code changes and keep tests up-to-date.

Of the three automation levels—UI, service, and unit—the unit level appears to be getting the least amount of attention. In fact, the attention level seems to be the inverse of what you would expect from a community following automation best practices like the testing pyramid.[8] For example, after taking a quick product offering survey in the space, I found only one commercial product that uses AI for unit test generation. This pales in comparison to more than 10 for functional UI test automation and 5 for API testing. However, AI technology to support improvements to automated unit testing is definitely available, and I hope to see more progress in this area in the near future.

AI for Performance Testing

Tooling for performance test automation has been relatively stable for many years. Several organizations still follow legacy load-testing practices that have a steep learning curve and involve a lot of manual steps. AI and ML are propelling us into a future where you can

5 Laurence Saes, "Unit Test Generation Using Machine Learning" (master's thesis, Universiteit van Amsterdam, August 18, 2018).

6 Diffblue

7 "GPT-3 Powers the Next Generation of Apps," OpenAI, March 25, 2021.

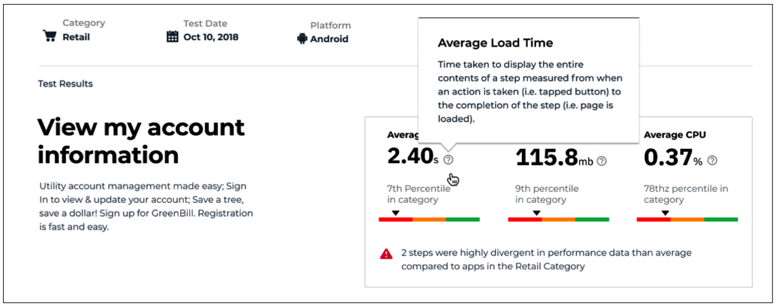
8 Mike Cohn, "The Forgotten Layer of the Test Automation Pyramid."

rapidly gather and correlate performance test metrics and even generate complete end-to-end performance tests that normally would require human experts. In this section, I summarize two promising directions in the use of AI for performance testing.

Application Performance Benchmarking

With an AI-driven framework for functional UI automation in place, you can extend the bots' capability to tracking key application performance metrics. While the bots are exploring and testing, they collect data on the number of steps taken, load times, CPU utilization, and more. With AI, not only can you see how your app performs in key scenarios, but you can also compare its performance to similar applications from your competitors once the app is available, for example, in an app store. This is possible because AI-driven tests are highly reusable across different apps within the same domain.

Figure 2-4 provides a sample report that compares the test results of a retail application with those of other applications in the domain. The way this works is that the bots run a set of goal-based tests on key use case scenarios for each app within a given category. The bots compute a percentile rank for each performance result so that you can compare the results for your app with all other apps in your category.


Category
Retail

Test Date
Oct 10, 2018

Platform
Android

Test Results

View my account information

Utility account management made easy; Sign In to view & update your account; Save a tree, save a dollar! Sign up for GreenBill. Registration is fast and easy.

Average Load Time

Time taken to display the entire contents of a step measured from when an action is taken (i.e. tapped button) to the completion of the step (i.e. page is loaded).

Averag
2.40s
7th Percentile in category

115.8mb
9th percentile in category

Average CPU
0.37%
78thz percentile in category

2 steps were highly divergent in performance data than average compared to apps in the Retail Category

Figure 2-4. Sample application benchmarking test report for a major app in the retail category

Toward End-to-End Performance Testing with AI

Although practically useful, application performance benchmarking is not the same as end-to-end performance testing. Recall from "Nonfunctional Test Automation" on page 6 that system-level performance testing emulates production traffic using hundreds and thousands of virtual concurrent users conducting real-world transactions. In a presentation at the STAREAST 2020 testing conference, performance testing expert Kaushal Dalvi shared his team's experience building a tool to generate end-to-end performance tests.[9] Figure 2-5 depicts a vision of their ambitious goal of developing a self-service system that automatically produces LoadRunner scripts complete with parameterization, correlation, and logic.

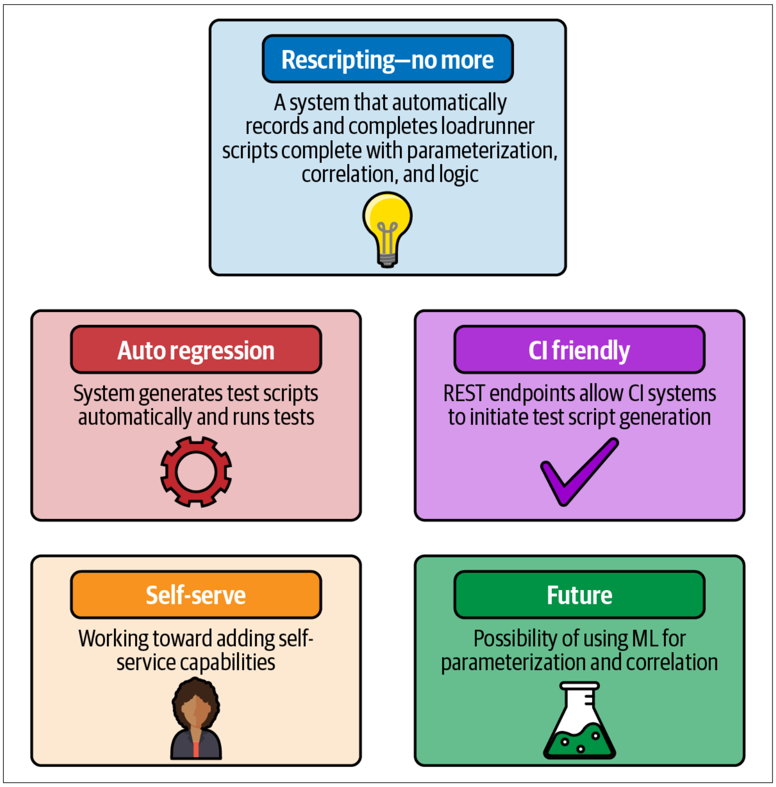

Rescripting—no more
A system that automatically records and completes loadrunner scripts complete with parameterization, correlation, and logic
Auto regression
System generates test scripts automatically and runs tests
CI friendly
REST endpoints allow CI systems to initiate test script generation
Self-serve
Working toward adding self-service capabilities
Future
Possibility of using ML for parameterization and correlation

Figure 2-5. A vision of automated end-to-end performance testing using ML[10]

9 Kaushal Dalvi, "End to End Performance Testing—Automated!" (paper presented at the STAREAST 2020 Conference, Orlando, Florida, May 2020).

10 Kaushal Dalvi, "End to End Performance Testing—Automated!"

The internal tool eliminates the need for manual rescripting, and now there is ongoing work that uses ML to drive smart rules for parameterization and correlation. Several application performance-monitoring vendors are also touting features that include AI-based performance, scalability, and resiliency testing.

AI for Design Testing

A quick internet search on the topics of manual and automated testing is likely to return some blog posts, articles, and presentations describing why test automation will never replace manual testing. One of the frequently used points to support this argument is that it is not possible to automate things that require human judgment. Such qualitative assessments rely on people's experiences, perceptions, opinions, or feelings. UI design quality attributes like usability, accessibility, and trustworthiness all fall under this category. However, advances in ML are demonstrating that it is possible for machines to simulate human judgment for specific tasks, including UI design testing.

AI for Mobile Design Testing

Google and Apple publish guidelines to help developers ensure that Android and iOS mobile apps are well designed and deliver a consistent user experience. Here is an example guideline:

When Possible, Present Choices
: Make data entry as efficient as possible. Consider using a picker or table instead of a text field, for example, because it's easier to choose from a list of predefined options than to type a response.

You've probably already noticed that, although these guidelines are written with good intentions, parts of them are vague and open to interpretation. For example, how efficient is "as efficient as possible"? Is there a rule to know when you can stop? What about widgets other than text fields? While this may seem like nitpicking, the subjective nature of the guidance and the resulting designs is what makes this problem so difficult. Furthermore, even when the guidelines are clear and precise, there are so many variants to check that doing so in an application-specific way is extremely tedious.

AI is a great way to catch these issues because you can train the bots to examine the screen just like a designer, customer, or reviewer.

They don't look at code or have app-specific checks but instead check all the visual elements on the screen against an AI trained on example guideline violations labeled by humans. They find these issues almost instantly and in a repeatable manner that avoids the error of human memory and interpretation. With AI enabling the automatic validation of UI design guidelines, there really is little reason for humans to look for the issues that machines can now identify.

AI for Web Accessibility Testing

In an effort to promote universal access to web technologies, the World Wide Web Consortium (W3C) developed a set of Web Content Accessibility Guidelines (WCAG). These guidelines provide criteria to make software accessible to people with physical, visual, cognitive, and learning disabilities. Not only do web development companies have a moral obligation to construct web applications that provide universal access, but in most countries they have a legal obligation. Although several tools support static WCAG web page analysis, present tools fall short in evaluating an entire application for accessibility. Furthermore, current test automation techniques are capable of discovering only about 30% of WCAG Level A and Level AA conformance issues.[11]

AI is proving to be an effective way to extend the capabilities of current accessibility-testing tools. By combining an AI-driven testing platform with open source tools, you can train the bots to explore a website and evaluate its WCAG compliance. As the bots explore the site, they conduct static accessibility checks using the open source tools and generate dynamic tests that mimic users with disabilities. An interesting project, code-named Agent A11y,[12] that employs this approach appears in the 2019 proceedings of the Pacific Northwest Software Quality Conference. A notable feature of Agent A11y is that, due to the large set of WCAG checks the bots perform, the authors even use ML to correlate and coalesce the accessibility test results. Talk about turning a problem on itself!

11 Aleksander Bai, Heidi Mork, and Viktoria Stray, "A Cost-Benefit Analysis of Accessibility Testing in Agile Software Development Results from a Multiple Case Study," *International Journal on Advances in Software* 10, nos. 1–2 (2017): 96–107.

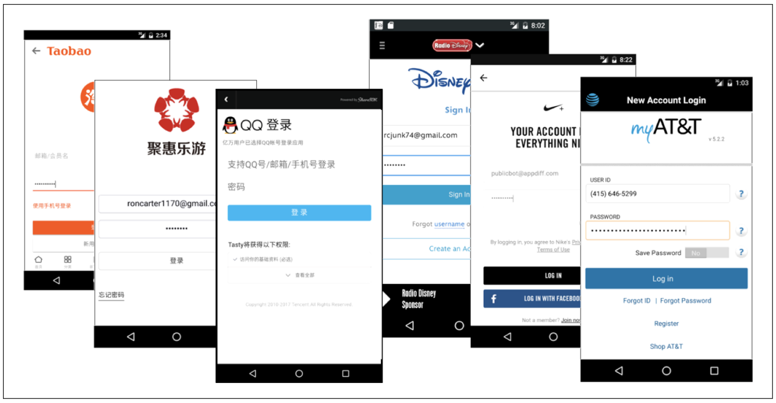
12 Keith Briggs et al., "Semi-Autonomous, Site-Wide A11Y Testing Using an Intelligent Agent," PNSQC Proceedings, 2019.

AI for UI Trustworthiness Testing

While coteaching a testing workshop, I had the opportunity to engage the participants in playing an intriguing game of human testers versus AI bots. If you think about it, AI can beat humans in games like chess, Jeopardy, and go, so why not software testing? Let's take a look at this game of testing and how it played out.

Envision 70 testers in a classroom. However, these testers are no ordinary testers. These are professional, technical testers, who work in roles where their company has chosen to send them for a week of training at an international testing conference held in the United States. These testers are confident enough to brave a full day of learning about AI and ML algorithms. This room is full of great testers. Their opponent is a neural network—AI bots trained on data related to the questions that are about to come.

Now let's go a step further and pretend that you are one of those testers and see if AI can beat you at your own game. We ask you the following qualitative testing question: If you were looking at an application's login screen, how would you know if you could trust it or not? In other words, solely by looking at the user interface, could you rate an app's trustworthiness? Take a moment to think about it and then look at some of the example mobile login screens in Figure 2-6.



roncarter1170@gmail.c

rcjunk74@gmail.com

publicbot@appdiff.com

(415) 646-5299

Figure 2-6. *Rating the trustworthiness of an application based on its UI design*

In Figure 2-6, the screens on the left are data samples of some of the least trusted apps, while those on the right are some of the most trusted. Any thoughts? If it's any consolation, the other 69 testers in the room are taking quite some time to think about it too. There are no quick answers. A woman in the front row exclaims, "Foreign languages!" She explains that if the primary region of the app store is the US, but the app is written in a foreign language, she wouldn't trust it because she wouldn't understand what it was saying. Not a bad start, but we've already spent three minutes with 70 human minds thinking about this problem in parallel. Now granted, it is not typical that you would engage 70 testers on this one problem, and there may be several biases at play here. However, even if only one highly skilled tester produced the same result, it may not be a good use of their time.

A couple more minutes go by, and then a hand goes up. A gentleman suggests that if there is a recognizable brand or logo on the screen, he would probably trust the app more than apps without these features. So now, 70 people have spent five minutes of their time, and we have two ideas for how to measure UI trustworthiness. That's progress, but the room quickly becomes quiet again. This is the point where the law of diminishing returns sets in. There are no more ideas past the 10-minute mark, and in fact there are no more ideas until that group has to move on.

But how did the AI bots perform in the challenge? Prior to the class, back at my company's headquarters, ML engineers trained a neural network using trustworthiness data from real users. The data was the result of asking individuals to rate the trustworthiness of a large set of login screens on a scale of 1 to 10. Once trained, the bots had the ability to simulate the human raters while working on previously unseen samples. As part of the experiment, the engineers inspected the neural network to understand the AI's answer to the same question I asked the human testers. Here is the AI's explanation:

Foreign language
: If the screen has foreign words/characters, it's less trustworthy.

Brand recognition
: If the screen has a popular brand image, it's more trustworthy.

Number of elements
: If the screen has a high number of elements, it's less trustworthy.

Interestingly, for this task, the AI appears to be “smarter” than any one person. The bots produce three UI design factors that relate to UI trustworthiness, whereas no single person came up with more than one factor. Furthermore, not only did the AI discover an additional aspect of the application UI that correlates to trustworthiness, but it gave a precise score of how trustworthy it thought the screen was using a scale of 1 to 10.

With machine learning, the bots truly learn to simulate the judgment of the humans that provide training data for a given task. In this testing experiment, the bots directly reflect how real users view trustworthiness, while as human testers we have to pretend and emulate that empathy indirectly. How much better is it to have the oracle be real-world users versus testers trying to reverse engineer and guess what the end user will think or feel?

Conclusion

AI-driven test automation is causing quite a stir in the software community due to its applicability to multiple levels and dimensions of software testing. AI is testing user interfaces, services, and lower-level components and evaluating the functionality, performance, design, accessibility, and trustworthiness of applications. With all of the activity and buzz around AI for software testing, it feels like the beginning of a new era of test automation. AI is giving testing some much-needed superpowers to help tackle challenges like automatic test generation.

CHAPTER 3

Automating Graphically Intensive Apps

Recall from Chapter 1 that one of the drawbacks of traditional UI testing tools is the lack of support for automating highly graphical applications. In this chapter, you'll catch a few glimpses of how AI is pushing test automation into uncharted territories—video streaming and game testing.

AI for Video Stream Testing

Whether watching a movie or show on demand on Netflix, getting live match updates on NBA.com, logging on to a Zoom meeting for work, or playing games via PlayStation Now, people are becoming more and more dependent on video-streaming technologies for business and entertainment. Video streaming is now a core experience in many familiar applications and next-generation gaming systems. Automating the testing of video streams is quite challenging. In this section, you'll learn about the current state of the practice and then dive into how AI is improving the level of automation.

Video Stream Testing Practices

Best practices for testing video stream quality today involve asking humans to watch videos and rate them using a *mean opinion score* (MOS). The MOS is a value in the range of 1 to 5, where 1 is the lowest perceived quality and 5 is the highest perceived quality. Not only is this manual approach an expensive and slow way to test video streaming quality, but it doesn't scale. When network engineers at streaming companies make changes to networks, tweak their decoders, or switch to new devices or carriers, they often can't do a full manual pass at testing the new or modified configurations. The traffic on the network, the variety of devices and even the types of videos make it difficult to measure the user experience.

Traditional test automation approaches for video quality involve code that combs through every frame of a video during playback, under various network, device, and encoder conditions, and tries to detect known types of failure modes through the image pixel patterns. Such approaches fall short in several ways, and their impact on the end user is difficult to quantify. The failure modes of video streaming range from simple blank screens to horizontal vertical lines to freezing frames, low frame rates, and more. But knowing how all these functional issues detract from the user experience is difficult. What if the issues happen during the credits versus the core of an action sequence? What if multiple failure modes occur at the same time? Automation today can find some of the issues, but it can't map them back to the true oracle—the end user.

Subjective measures of quality are made even more difficult because videos can be riddled with issues that go unnoticed because they are in noisy or less relevant areas of the screen. For example, Figure 3-1 shows a scene in a television series with people talking. During a given scene, these videos may have issues in the furniture, windows, props, or even people in the background. However, because the human eye is likely focused on the areas of the video where the subjects are speaking in the foreground, the background issues may not be noticed at all and the video segment is given a perfect 10 out of 10. To be great at assessing video quality, test automation would have to simulate all of this!

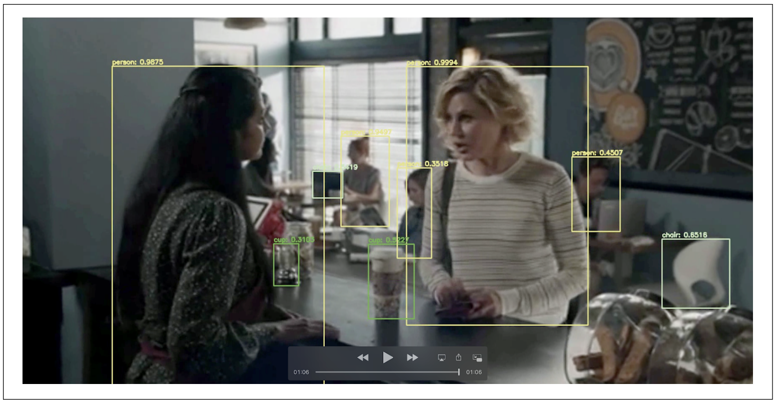


Figure 3-1. AI bots analyze a television scene for interesting objects during video quality testing

Automating Video Quality Testing with AI

Can you already see how to leverage AI and ML for automating video stream testing? By now you've probably noticed two distinct patterns in AI-driven testing solutions. The first is that you use computer vision to give the bots "eyes" like humans. Next, you transform the aspects of the problem that require human judgment in ML training exercises. Here's what an AI-based approach for video stream test automation looks like:

1. Gather training data in the form of video clips and their associated human MOS ratings.
2. Train AI bots to analyze the video clips, including teaching them to recognize the following:
    a. Interesting or attention-grabbing objects in a scene
    b. Video quality glitches within the bounds of the interesting objects
3. Train the bots to map all of the above video outputs to corresponding MOS values.
4. Leverage the bots to predict the video quality for streams it has not previously seen.

By creating a set of internal video benchmarks and working with crowd-sourced human judges, my R&D team at test.ai was able to use the aforementioned approach to implement a video quality MOS predictor. Although their first model was naive, the training results were relatively good, with a 65% correlation between the AI-predicted scores and the human MOS values. Note that correlations of 70% or higher are considered strong. However, after enhancing the naive model with features that accounted for video categories and scenes, they saw the correlation rise to 78% and 89%, respectively. Being perfectionist data nerds, they worked further to train the model on larger, independent, academic datasets of videos with MOS scores.[1] Figure 3-2 shows the results of one of the best-performing categories, Asian Fusion, which yielded a correlation of 93%.

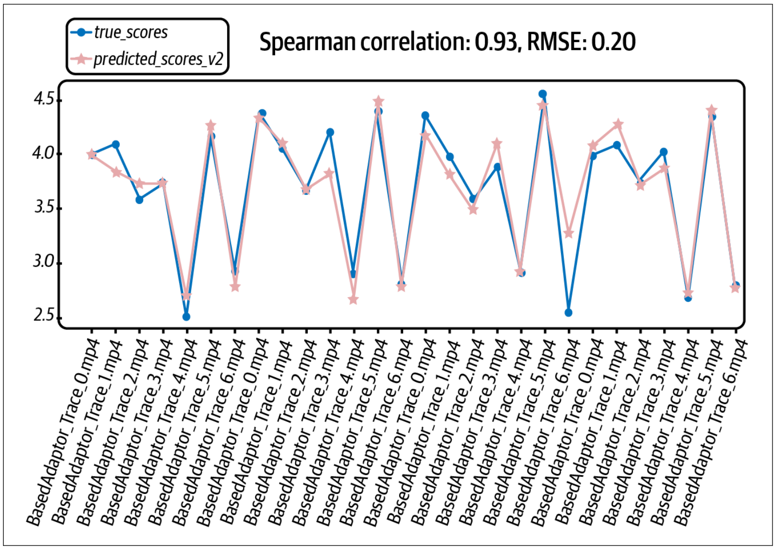


Figure 3-2. AI predictions versus human ratings on video quality for the Asian Fusion category

With AI on its way to replicating human MOS judgment scores, perhaps engineers developing video-streaming solutions will soon be able to get near real-time estimates of video quality that are just as good as human ratings. Although there is more work to be done, the

1 "Image and Video Quality Assessment at LIVE," Laboratory for Image & Video Engineering, The University of Texas at Austin, accessed June 3, 2021.

progress so far in this area is promising and serves as a good example of how to apply AI to today's test automation challenges.

AI for Game Testing

Testing modern video games is no small feat! Today's video games are highly visual, graphical, customizable, updatable, streamable, and cross-platform compatible. Game testers must cover multiple dimensions of an uncountably infinite state space in a constantly evolving application. These dimensions include testing the functionality of the gaming application itself, visual aesthetics of graphics and animations, and trying to break the gameplay mechanics.

AI has been playing games for as long as we can probably remember, so why not test them too? Let's investigate the exciting world of video game testing as a final look at how AI is bridging the automation gap.

Video Game–Testing Practices

Writing automation to test gameplay is extremely challenging. Any approach that hardcodes assumptions will constantly break as the underlying game engine and its elements are usually in flux. The content, characters, items, maps, and storylines of games are continually evolving. Some games even let the user change these elements in real time as they play. Software development toolkits for games generally come with a low-level internal test-scripting language. This scripting language enables developers to write unit tests for verifying core game logic, and that layer of the application tends to be quite stable. However, the language lacks the ability to drive the user interface and runtime rendering. As a result, automated testing at this level cannot catch the issues that real-world players encounter during gameplay or when navigating the gaming application.

Testing Game Applications with AI

With a lack of automated tool support for testing video games end to end, you'll find that many gaming companies do not even have smoke or sanity tests in place for their gaming applications. A *smoke or sanity test* is a very basic test that helps you to determine whether a new version of the application is stable enough for further testing. For example, a video game smoke test may launch the game app, navigate through the menus to start a new game session, and interact with the gameplay elements once the game has loaded. Recall from "Limitations of Traditional Approaches" on page 4 that the reason there is a lack of support is that most functional UI automation tools require applications to have a DOM. However, due to their highly visual and graphical nature, many parts of gaming applications lack such a model. Fortunately, AI-powered testing tools are able to use computer vision to interact with the game menu icons and other widgets to make this type of automation possible. Furthermore, AI for game application testing extends beyond smoke tests, as this technology is enabling gaming companies to define complete functional testing scenarios for locally hosted games and those streamed from remote servers.

Testing Game Stores with AI

Many video games now have an in-game store that allows players to purchase add-ons in the form of downloadable content. Although you can view this type of testing as an extension of functional UI testing, it raises some unique testing challenges that make it an apt illustrative example. Items in the game store's UI tend to be animated, constantly changing position and orientation on the screen. They may also appear on the screen in several different orderings as a player can pick any given item at a moment in time. Figure 3-3 provides three screenshots taken of an axe from the game store UI of a popular game. In the menu, this particular axe rotates around the y-axis. As you can see from the illustration, as the axe rotates, it becomes more difficult to recognize that it is the same axe or even an axe at all. Now imagine that this axe glows, pulses, and sparkles periodically as it rotates. When games render objects like this at runtime with varying visual effects, traditional image-seeking or pixel-scanning automation techniques fail to identify them correctly.

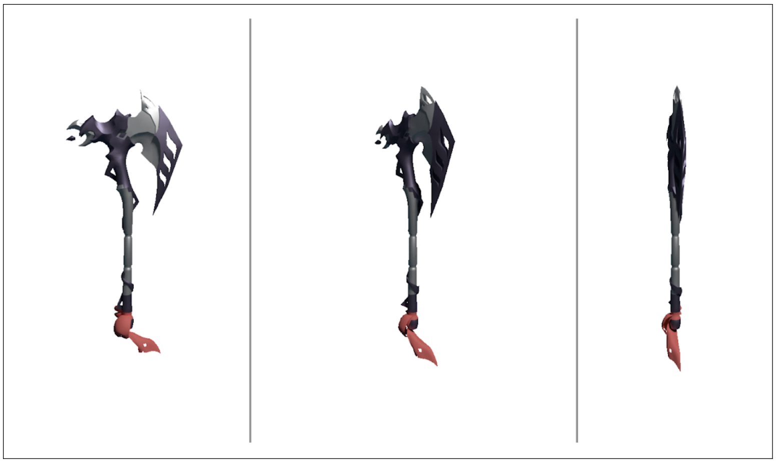


Figure 3-3. An add-on weapon rotating in the game store menu of a popular title

On the other hand, AI-based solutions work great to classify objects under these types of circumstances. You can train the AI using images captured from a video of the animated axe and, just like a human, the machine is able to recognize the axe from almost any angle or in combination with different types of lighting and artistic effects. AI therefore makes it possible to identify dynamically rendered items in video games—a feat that foils conventional test automation approaches. This testing capability is particularly important for in-game stores since they have a direct revenue stream. However, being able to classify these game assets is just the first step. Next, we'll learn how AI verifies game assets to make sure that gamers aren't surprised by cosmetic bugs after purchasing items from the store.

Testing Gaming Assets with AI

Figure 3-4 illustrates one approach to automatically detecting cosmetic bugs in gaming assets with AI. The approach involves training ML models to visually identify and classify each type of visual asset (e.g., character model, weapon, etc.) on the screen and compare these ground truth images to visuals from subsequent software releases. These are the key steps of the approach as they relate to Figure 3-4:

a. Visually isolate the baseline asset that represents the ground truth by removing the background. You can do this by training an ML model yourself or using a third-party tool that does the work for you.

b. On the new version of the software, run a test that uses image-based UI element selection to navigate to the asset in the game store and visually capture the test observation. You will have to run background removal on the test observation since the bot will capture the raw image of the asset from the game.

c. Generate an image mask highlighting the differences between the baseline and test observation images. You can train an ML model to do this, but first you may have to read up on how to compare two images using OpenCV and Python.[2]

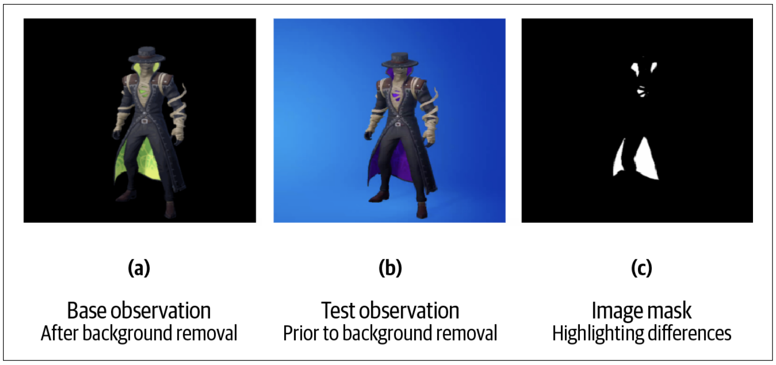


Figure 3-4. Detecting visual differences in game assets during AI-driven test automation

Aside from visual game asset verification, you may also want to test the audio assets of your game. For audio, let's look at a different but equally powerful way to carry out the task of comparing gaming assets during testing. In this case, instead of trying to detect differences, you can frame the audio verification problem as one where you are looking for similarities between a baseline audio file and one observed during testing. Figure 3-5 illustrates one way to verify gaming sound effects. It shows three *spectrograms*, which are graphical representations of the amplitude and frequency of a sound wave

2 Adrian Rosebrock, "Image Difference with OpenCV and Python," PyImageSearch, June 19, 2017.

over time. Spectrogram A on the left of Figure 3-5 is the baseline sound of an axe being swung, while Spectrogram B on the far right is the sound of the same axe being swung in a subsequent build of the game. The approach to automatically testing these sound effects is to compute the cross-correlation, also called a *sliding dot product*, represented by Spectrogram C in the middle of the figure. The general idea is that, accounting for any differences in amplitude and time, the model keeps one signal fixed and determines if the other signal sweeps through it.

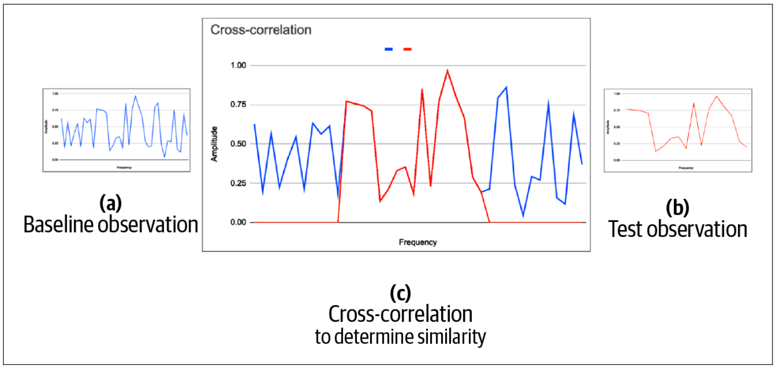


Figure 3-5. Testing sound effects in video games by using cross-correlation to determine similarity

Now that I've walked you through two asset verification approaches, one for visual assets and the other for audio, you should have a good understanding of how this technology can be applied to game testing. Although at times it feels like magic, AI and subfields like computer vision are rooted in mathematics, computation, logic, and other disciplines including philosophy, psychology, biology, and evolution.[3] As such, there are several other ways to leverage AI to tackle these problems.

3 John A. Bullinaria, "IAI: The Roots, Goals, and Sub-Fields of AI," University of Birmingham, 2005.

Gameplay Testing with AI

Machines have been playing games for as long as I can remember, from brute force approaches that exhaustively search the state space of games like tic-tac-toe and chess to the more recent deep ML techniques capable of playing online, cooperative, multiplayer games like *Dota 2*.[4] Neural networks like Open AI Five leverage reinforcement learning at a large scale to train bots that play together as a team, and they are able to win back-to-back games against *Dota* world champions. Bringing together everything you've learned so far about functional UI and game asset testing with AI and combining it with the ability for AI to reach expert levels of gameplay via self-play and reinforcement yields a powerful approach to testing modern video games.

Figure 3-6 provides a snapshot of this approach in action, which involves describing test steps as high-level goals for an AI bot, and then, through the process of reinforcement learning, the AI is left to play the game until it achieves the goal. A test for an online shooting game may be to explore a location; collect weapons, ammo, and equipment upgrades along the way; and then score a kill. While the bot is autonomously executing the gameplay steps from the test, it is also checking for cosmetic bugs and audio glitches. If the bot is able to accomplish the tasks without discovering any bugs, then you know that those features are working as intended for the given scenario—all without human intervention. Furthermore, you can give the bot goals that seek to intentionally explore the boundaries of the gameplay environment, or you can reward it for actions such as jumping on walls and ceilings, in hopes of breaking the game physics. As the application evolves and gameplay changes, your AI-based testing approach autonomously relearns how to accomplish the same tasks in the new environment.

4 "OpenAI Five: 2016–2019," OpenAI, 2019.

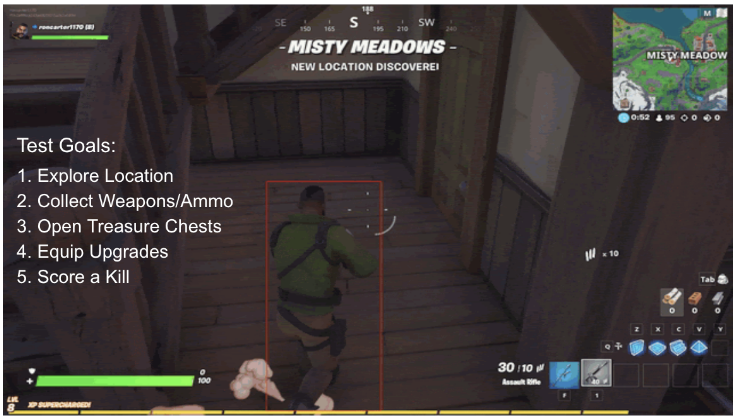


Figure 3-6. Gameplay testing using goal-based, reinforcement learning and asset verification

Conclusion

AI is enabling the automated user experience testing of graphically intensive, multimedia applications, an area that conventional tools still do not support well. Video streaming and gaming are examples of two industries where this technology is proving to be a game changer, no pun intended. However, several additional opportunities exist for automating multimedia applications. Computer-aided design (CAD) software used in the automotive, shipbuilding, aerospace, architecture, and prosthetics industries, among others, is eagerly awaiting better automated testing solutions. I truly hope that, as with video streaming and gaming, AI will soon be giving test automation in those areas the “eyes” and “ears” it needs to succeed.

CHAPTER 4
Costs and Benefits of AI-Driven Testing

Throughout my career, I've worn quite a few different hats on AI-driven testing projects, so I'm going to share a bit about the costs associated with these types of projects. As a test architect, I worked on a team developing a homegrown, internal tool for AI-driven test automation. The project had a lofty goal to reduce the manual effort associated with testing by 25%. Notable AI-driven features of the tool included automatic exploration and visual state modeling of the application under test; test generation using well-known techniques such as boundary testing and equivalence partitioning; goal-based testing and coverage analysis based on model-based testing criteria; and the extraction of test information from user stories. Later, as an engineering director, I was the key client stakeholder responsible for the testing teams that would be consuming this solution on a daily business. Currently, in my role as a chief scientist, I lead research and development of a commercial AI-driven test automation platform.

In this chapter, you'll learn some of the benefits of AI-driven test automation in practice. I'll then give you a holistic view of what the cost-benefit analysis looks like from a business perspective. Although the information at this level does not provide you with specific numbers for investment costs and savings, it should give you a solid foundation for understanding the key factors associated with the return on investment (ROI) of AI-driven test automation.

Investment Costs

Based on my experiences in AI-driven testing projects, I'll present a well-rounded view of AI-driven testing investment costs, including the up-front and recurring costs you can expect if developing and maintaining a homegrown solution and what's involved in purchasing a commercial, vendor-based solution.

Homegrown Solutions

If you decide to build an AI-driven test automation platform or framework in-house, you'll first need the talent to do so. Since labor tends to be the highest recurring cost in software development, you'll want to pay close attention to these requirements before you go any further down this path. You'll need a team of folks who have deep knowledge and experience in software testing, software engineering, machine learning, research, and IT operations. Ideally, you'll want individuals who have a cross section of these skills. You'll also need a strong engineering leader who can drive the vision, track the project, and work with key stakeholders in the organization throughout development, alpha and beta testing, and adoption.

As you can imagine, the talent required for this type of project is difficult to find and does not come cheap. When I was part of such a team, there were approximately 10 people, 3 of whom had doctoral degrees in computer science focused on software testing and the rest with extensive experience in computer vision, natural language processing, and AI-related disciplines. Those who didn't have knowledge or background in a given area generally spent a portion of their time every week doing self-paced learning or getting nanodegrees to bring themselves up to speed. Fortunately, some of the talent already existed in the organization, but it was also necessary to recruit for some of the expert positions. Labor cost for the relatively small team was over $2M per year, and it was two years before the first alpha was released. Much of the initial time was spent on gathering and curating training data, performing research tasks, and iterating between research and product development.

Up-front costs included capital expenses related to the project's hardware and software requirements, the highest of which was deep learning servers for training ML models locally. Alternatively, you could use on-demand, CPU, and GPU cloud compute services.

However, with so much initial research and experimentation, investing in the servers up front will typically be cheaper in the long run.

After the initial rollout of your homegrown solution, you'll still have to make significant investments in its development, evolution, and maintenance. As a result, you are likely to always have an engineering team supporting it, as well as a research team to investigate how the latest advances in AI/ML can help further the benefits of the framework or platform.

Vendor-Based Solutions

Over the past decade, there has been significant growth in the number of vendors offering AI-powered testing solutions. Figure 4-1 provides a snapshot of some of the vendors in this space. Which vendor or solution you go with really depends on your needs, level of investment, and expectations for the ROI. I always recommend starting your search with a set of criteria that describes the capabilities and partnership values that are important to you. Yes, "partnership." In my experience, even if an available solution appears to fit your needs off the shelf, since many of these tools and frameworks are still in their infancy, you will want to ensure you're choosing a good partner to guide your organization through integration and adoption.

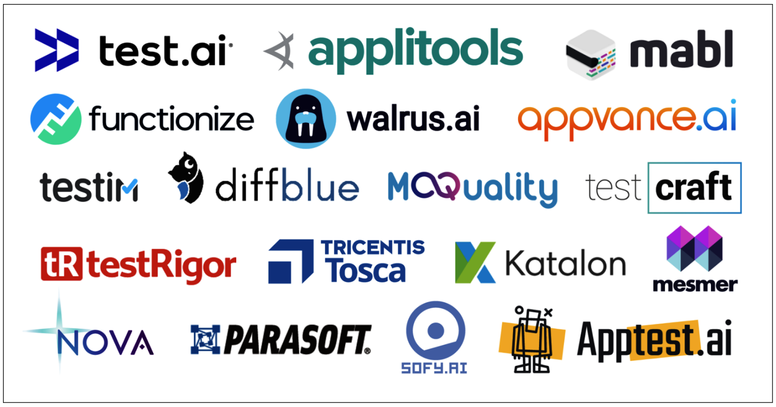


Figure 4-1. Vendors that offer AI-driven automated testing solutions

If you do end up going with a vendor-based solution, the most obvious cost is a software license fee. Depending on the product, this may be a one-time cost, but it is more likely to take the form of a recurring software subscription. Furthermore, if you think that your organization has unique test automation challenges that AI can solve, set aside some of your budget for professional services and engage the vendor in developing custom modules for you. This is yet another reason why it is so important to select the right partner. Remember, what you're investing in here is more than the solution; you're also making an investment in the vendor's ability to support, maintain, and evolve it. While this may be true for many of your vendor engagements, it is especially true in the rapidly changing world of AI-driven test automation.

ROI

Regardless of whether you build or buy, prior to investing in this technology you'll want to understand its ROI. So what are some reasonable and realistic benefits from implementing AI-driven test automation? Let's discuss some of the practical benefits you can expect to see and then take a look at the big picture of how the investment can positively impact your organization's total cost of testing.

Practical Benefits

Automated testing with AI brings several practical benefits to development teams, including increased coverage, acceleration, reuse, scalability, robustness, and resiliency. These shouldn't be completely new to you since I've already touched on some of these when describing AI for testing approaches in Chapters 2 and 3. However, here I'll expand on each benefit, adding a bit more detail and context on the ROI that organizations typically realize when applying AI-driven test automation in practice.

Increased Coverage and Acceleration

Grand testing challenges such as input and state explosion are some of the key reasons why coverage is a central theme in software testing. When new features are added to an application, its complexity increases exponentially due to the interactions of these new features with existing components.[1] However, traditional approaches to test

1 Jason Arbon, "AI and Machine Learning for Testers Jason Arbon, Appdiff," 35th Annual Pacific NW Software Quality Conference, October 18, 2017, YouTube video, 44:16.

automation involve adding handcrafted test cases one at a time. As shown in Figure 4-2, over time the test coverage required to validate the quality of your software product diverges from the engineering team's ability to design and write test scripts for it.

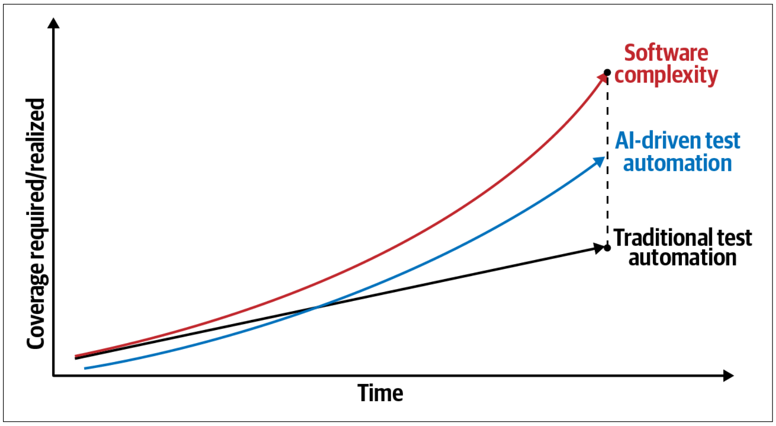


Figure 4-2. Increasing test coverage with AI-driven test automation

Automated testing with AI can increase both the level of test coverage and the speed of testing. As shown in Figure 4-2, accelerated coverage essentially narrows the coverage gap between software complexity and test automation. Recall from "AI for UI Testing" on page 9 that you can train bots to generate test inputs and expected outcomes automatically. Furthermore, in Chapter 3, you saw that test scripts for video games can be specified as high-level goals, which the bots then use to autonomously explore and search for bugs while trying to achieve those goals. AI therefore reduces the manual effort for creating tests, allowing you to reach higher levels of coverage. Combining AI-driven test generation with other benefits like reuse, scalability, robustness, and resiliency allows testing to keep pace with continuous delivery and the ever-growing demand for more features. Accelerated test coverage translates into a faster time to market and shorter cycles for reproducing and fixing bugs. This in turn can lead to increases in customer satisfaction, retention, net promoter scores, and sales.

Reuse and Scalability

AI-driven test generation systems work in a very general and generic way.[2] As a result, the same test generation approach can often be applied to multiple software applications or components. Furthermore, at the user interface level, leveraging computer vision and goal-based reinforcement learning allows for the same test to be run against multiple applications within a given domain. For example, the retail shopping apps from Amazon, Walmart, and Target can all be tested by a bot following this high-level test case flow: "login," "add items to cart," and "checkout." Recall that these types of reusable, abstract test cases are the same ones that enabled the bots to gather insights during "Application Performance Benchmarking" on page 17 and "Video Game–Testing Practices" on page 29.

With AI generating tests automatically, you just might need an army of bots to run all of your tests. Each of these bots can consume a significant amount of resources, especially if they require deep learning models. The good news is that cloud compute is readily available and accessible. Leveraging a distributed architecture can enable you to efficiently and reliably scale to support the needs of large enterprises.[3] All of the major players like Google, Amazon, and Microsoft offer cloud services for training and even autotuning ML models. Outside of ML training, many cloud providers and open source container-orchestration systems have features such as autoscaling, where the infrastructure dynamically adds or removes nodes based on demand and/or utilization. Chances are that your organization is already plugged into one of these ecosystems, and if it's not, it probably soon will be.

Robustness and Resiliency

AI-driven test automation tends to be more robust and resilient than traditional test automation. Recall from "Limitations of Traditional Approaches" on page 4 that this is particularly true at the UI level, where conventional tools like Selenium and Appium locate screen elements using a DOM. Robustness in this context has to do with the ability of existing test scripts to withstand UI design

2 Dionny Santiago, "A Model-Based AI-Driven Test Generation System."

3 Patrick Alt, "Deploying a Large Scale Army of AI-Driven Testing Bots" (paper presented at the STAREAST 2021, virtual, April 2021).

changes. Consider an example where the frontend development team just performed a huge visual overhaul of your application. They've changed the look and feel of icons and graphics, moved around navigation bars and other widgets, and even introduced an entirely new UI framework. As such, the DOM of your application has drastically changed, and any test based on information from the previous DOM is now failing. I've lived this scenario many times and, in some cases, seen it delay or prevent organizations from providing a better user experience for their customers.

Now imagine that those same test scenarios are implemented using AI. First, they're not tied to the DOM and so aren't doomed to fail like the other tests. However, there were visual updates to the icons and graphics, and the AI-based tests are based on images. The good thing here is that even if you changed the shopping cart icon, you as a human can still recognize it. That's because you've probably seen hundreds of shopping carts in your lifetime. But guess what? You can train the AI on thousands of shopping cart icons, and, in practice when this is done well, the bots will recognize the updated visual no matter where it is located on the screen. Even if the bots failed to recognize it, the resulting test failure could be an indication that a human user might also experience difficulty recognizing it. After all, it doesn't look like any of the hundreds or thousands of examples from various applications, possibly including those of your competitors. Do you really want to be the only app design in the app store that users have a hard time understanding? In this regard, not only is the AI-driven automated test more robust, but it also provides more business value to the design and engineering teams because it mimics the way the end user interacts with your application.

If you're anything like me, you're probably already trying to find other ways the UI redesign may cause the AI to fail. Suppose the entire flow of the application changes? What if new screens are added or removed, or the icon is replaced by a link or some other type of widget? Surely, then, AI-based scripts will fail just like traditional automation, right? Well, not necessarily. This is where the AI bots' resiliency kicks in. Although the terms *robustness* and *resiliency* are sometimes used interchangeably when it comes to automation, they are slightly different concepts. Whereas robustness has to do with the ability to withstand changes, resiliency is more about adapting to changes. In the case where the entire flow of the application changes, remember that the bots leverage goal-based reinforcement learning to explore and model the application. As a result, if screens or widgets are completely removed or replaced, the bots will still try to complete their goals by trial and error. In the worst case, the bots have to completely rebuild their UI model of the application. However, since this happens automatically, it is likely to be way cheaper than paying highly skilled engineers to fix scripts with broken element locators.

Impact on Total Costs

Now that you have an understanding of the ROI from a practitioner's standpoint, let's take a look at what this all means for your organization's bottom line. More specifically, I'll answer the following question: how does the integration of AI-driven test automation impact total costs? A quick warning that I will get into a little bit of math here, but I promise to keep it relatively high level and not lead you down a rabbit hole of complex ROI formulas, estimations, or spreadsheets.

Let's begin with an organization's investment in software testing. By this I mean that, even without introducing AI-driven automation into the equation, every year an organization allocates a portion of its budget to software testing. This generally consists of both labor and assets. Software-testing labor includes any testers, developers, or business analysts who are designing, executing, or debugging test cases, filing and triaging bugs, and so on. Test assets range from hardware equipment and infrastructure to software licenses for testing tools and frameworks. For illustrative purposes, let T be an abstraction of the testing effort. You can then define the cost of testing as a linear function $f(T)$, represented graphically in Figure 4-3. The steepness or gradient of the slope of $f(T)$ is directly impacted by a constant c, which indicates the investment costs in testing labor and assets. Simply put, the larger your testing investment, the steeper the slope of the cost function and, inversely, the smaller the investment, the gentler its slope.

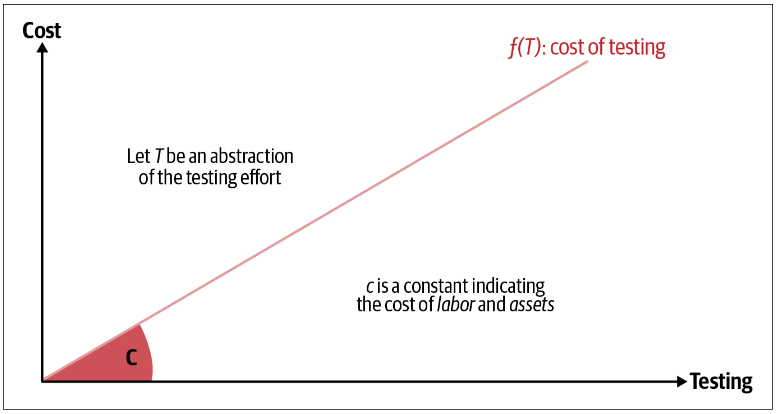


Figure 4-3. The cost of a testing investment T *is a linear function* f(T) = cT.

Unfortunately, when it comes to software testing, investment costs are just one side of the story. You see, even if an organization doesn't make an up-front investment in testing, it usually ends up "paying" for it in other ways. Do you know what I am referring to? It's the cost associated with not testing or not testing enough. When customers encounter high-severity defects in your product, or several low- to medium-severity defects for that matter, they may perceive the product as being low quality. Poor quality can lead to significant financial loss or even irreparable damage to the organization's brand and reputation. One of the value propositions of testing and test automation is identifying those issues early, prior to release, allows them to be fixed.

Figure 4-4 visualizes some of the various categories of defects based on whether they were found in development prior to release or post-release in production, along with any associated costs or risks. These are the categories, starting from left to right in Figure 4-4:

Known issues, won't fix
: Any reported issue found pre- or postrelease that the business has decided to not fix. These issues represent accepted risk.

Internally found and fixed in development
: Defects discovered by the engineering team prior to release and that the business has decided to fix.

Internally found in production and fixed
: Defects discovered by the engineering team after the release and that the business has taken action to fix.

Customer found and fixed
: Defects reported by the client or end user after the release and that the business has taken action to fix.

Undiscovered defects
: Issues that have been reported by neither the engineers nor customers, pre- or postrelease. These issues represent unknown risk.

The later the stage of the development cycle in which a defect is found, the more expensive it is to fix. As a result, any defects that escape to production pose a risk to the project. Those which are found in production, either by engineers or customers, may be viewed as cost inefficiencies. In other words, investing more up front in testing to find these defects earlier would potentially save the organization money. Furthermore, accelerating coverage through automation can also result in fewer defects escaping to production, which also reduces risk.

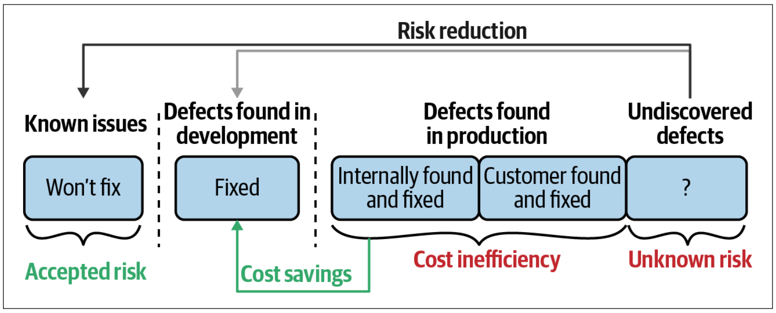


Figure 4-4. Various categories of defects and their associated risks and costs

So does this mean that you should pour all of your budget into testing and new, shiny test automation tools? Of course not. You see, when I was a young, inexperienced, and overzealous testing professional, I thought that testing and quality were nonnegotiable. Now that I'm older and a bit wiser, I've learned that quality, too, is negotiable if you want to make smart business decisions. However, to be able to do this in this context, you have to consider both the cost of testing and the cost of *not* testing at the same time.

In Figure 4-5, the function *D*(*T*) represents the cost of not testing, which is essentially the cost of escaped defects. This curve exhibits exponential decay, the opposite of exponential growth, and I've placed it on the same graph as *f*(*T*) so you can visualize them together. Looking at the graphs, you can see that when *f*(*T*) is extremely low, *D*(*T*) is high. In other words, when there is little or no investment in the testing effort, the organization incurs significant cost and risks due to escaped defects. However, as *T* increases, the costs and risks associated with escaped defects drop drastically. This is because when you first start testing a new product or feature, finding the first set of defects happens relatively quickly. However, as more and more testing is done, it can become harder and harder to find that next defect. Eventually, beyond where the two curves intersect, *T* can get so large that the law of diminishing returns sets in. Now you're essentially overtesting the product because although you're investing more in testing, you're not getting a good return on that investment.

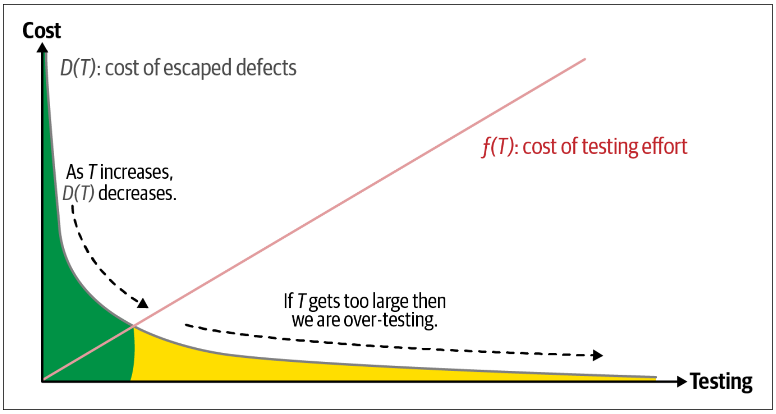


Figure 4-5. The cost of not testing is the cost of escaped defects D(T)

When considering how much to invest in testing and test automation, you need to consider the total cost of testing. Figure 4-6 introduces a new function *C*(*T*), the total cost of testing.

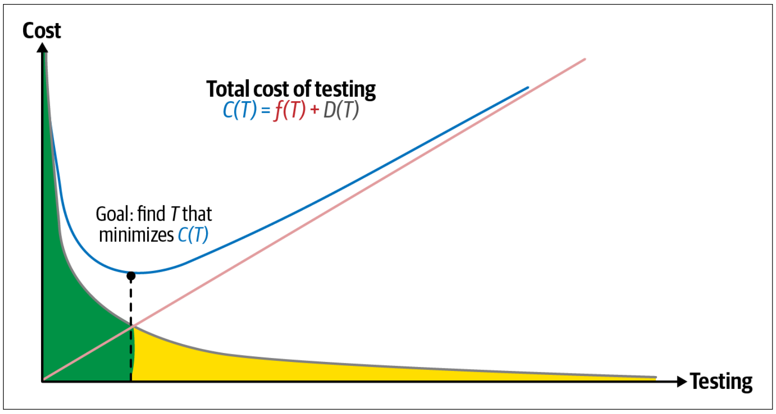


Figure 4-6. The total cost of testing C(T) *is the investment cost plus the cost of escaped defects*

As its name suggests, the total cost of testing *C*(*T*) is the sum of the testing investment *f*(*T*) and the costs of escaped defects *D*(*T*). The result is the blue, parabolic curve. The reason I mention its shape is because ideally you want to find the level of testing investment that minimizes the total cost of testing. In Figure 4-6, that's the lowest point on the not quite U-shaped parabola. Too little investment prior to this point means the organization is likely experiencing cost inefficiencies, while too much investment beyond it results in a negative ROI.

Now that you have a framework for understanding the key factors that impact your total testing costs, I can use it to finally answer the question set forward at the beginning of this section: how does AI-driven automation specifically impact total costs? Throughout this report, you have seen a recurring theme: AI-driven test automation is bridging the gaps between human-present and machine-driven testing capabilities. The end result is a significant reduction not only in labor costs but also in total costs.

Recall from Figure 4-3 that labor and asset costs indicate the value of the constant *c*, which determines the steepness of *f*(*T*). Since labor costs are almost always your biggest line item, investing in AI-driven test automation, or any process or tool that can act as a force multiplier for that matter, will lower the value of *c*. Figure 4-7 shows the ripple effect that this will have on the total cost of testing. As expected, reducing the cost of labor and assets produces a gentler slope for the investment cost function *f*(*T*). This results in the right side of the U-shaped total cost function $C(T)$ becoming flatter and more open. The minimization point for $C(T)$ moves down and to the right. In other words, when compared with the previous model, any given level of investment prior to reaching that minimal point now allows for a greater testing effort or a lower overall total cost. Furthermore, even if the team were to go beyond that minimal point into overtesting the product, it would still be cheaper to do so with machines rather than people, as indicated by the shaded area in Figure 4-7.

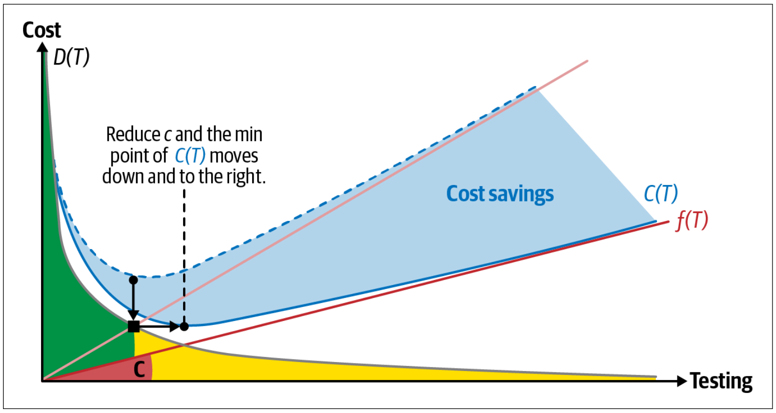


Figure 4-7. Reducing labor via AI-driven automation significantly reduces total testing costs

Conclusion

Determining how much to invest in AI-driven testing automation, or any advanced testing technology, should be based on a careful analysis of its ROI. My hope is that, through this chapter, I've provided you with enough insight into the available options and their respective costs and benefits so that you can make a more informed decision for your business. Whether you are an individual contributor or a leader in your organization, being able to formalize and articulate the cost and value of testing can be the difference between thriving and surviving.

CHAPTER 5

Future Directions

Now that you've taken the tour of AI-driven test automation, let me take you on a journey into the future of this emerging technology. In this chapter, I share my thoughts on how AI-driven test automation tools will evolve over the next decade. In my opinion, the AI for automation evolution and/or revolution, if you watch enough Hollywood movies, involves three steps. Within the context of software testing, these including using AI for the following:

1. Enhancing existing tools and frameworks at each testing level and dimension
2. Full stack replacement of entire test automation tool sets
3. Adaptive ML systems designed with self-testing capabilities

Although I have zero faith in my ability to predict the future, I do think it's important to spend some time discussing and theorizing about the future directions, paths, and intersections of AI and testing, prior to wrapping up this report.

Enhancing Existing Tools

Considering the current state of the art, I believe the immediate future of AI-driven test automation is on track with continued integration of AI into existing tools that target different testing levels, quality attributes, and application domains. As shown in Figure 5-1, the trend is likely to be one where researchers and practitioners continue to tackle each concern somewhat in isolation, until individual areas become stable and reliable.

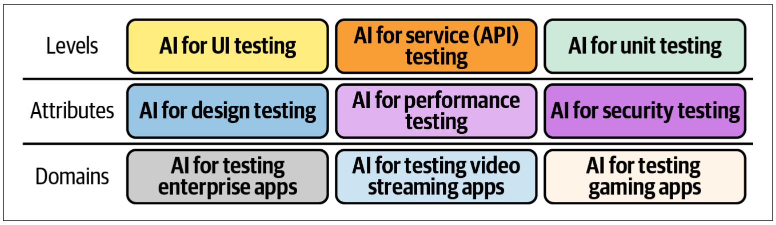

Levels
AI for UI testing
AI for service (API) testing
AI for unit testing
Attributes
AI for design testing
AI for performance testing
AI for security testing
Domains
AI for testing enterprise apps
AI for testing video streaming apps
AI for testing gaming apps

Figure 5-1. In the near term, AI enhances existing tools that tackle individual testing concerns

Full Stack Replacement

After its successful application and adoption in specific testing areas, I foresee AI-driven automation shifting toward tighter tool integrations and a more holistic testing solution. By building on past experiences, sharing training data, and cross-pollinating various techniques and lessons learned, practitioners will be able to develop general ML models and classifiers for automating all kinds of testing tasks. Figure 5-2 depicts the idea that in the medium term, AI replaces full stack testing with capabilities that address both core and cross-cutting concerns across multiple application domains.

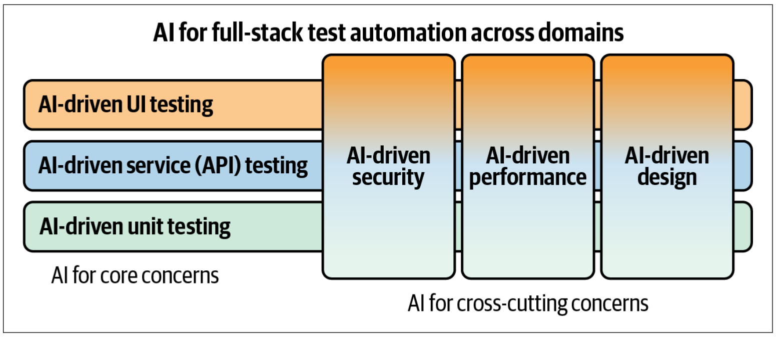


Figure 5-2. In the medium term, AI replaces full stack test automation across application domains

Initially, full stack solutions like the one shown in "Self-Testing Adaptive AI" on page 53 may still require regular human intervention for complex testing decisions. However, over time, as the AI trains on that human feedback and the technology advances, the bots will eventually take over those testing tasks.

Self-Testing Adaptive AI

Until now, I've focused your attention on what AI can do for testing and test automation. However, I believe that AI needs testing just as much as testing needs AI, if not more. Technology giants like Microsoft, IBM, and Google are all grappling with ethical, fairness, privacy, security, and a slew of other issues surrounding AI.[1] As more governments and businesses incorporate AI into their decision making, there is increasing risk that data biases and blind spots can lead to discrimination, financial loss, or even death. With so much on the line, it is important to bring a testing mindset and perspective to the world of AI. Even though an ML system may be complex, it can be quantified, measured, and controlled, not just with statistics but with thorough testing.

AI that incorporates both online and adaptive ML presents an interesting validation and verification challenge. Unlike offline ML systems that learn from well-organized datasets through single-batch processing, online ML leverages multiple data sources that are continuously delivering real-time data via sensors. These types of systems learn and adapt their behavior at runtime in response to environmental changes. Online ML is necessary in highly dynamic environments, where it is typically infeasible to stop the system, recollect data, retrain, and retest previously learned models whenever the environment changes.

Testing adaptive ML requires testing to occur online while the system is operating.[2] Figure 5-3 presents a long-term, future vision of AI. In that future, AI-based systems incorporate adaptive ML to keep pace with changes in user needs and application environments. A self-testing framework, which leverages AI for automated testing, is responsible for validating dynamic runtime adaptations. Now while the idea of self-testing in adaptive systems is not new, the technology is finally here to make a potential solution to these challenges a reality.

1 Tom Simonite, "Tech Giants Grapple with the Ethical Concerns Raised by the AI Boom," *MIT Technology Review Magazine*, March 30, 2017.

2 Tariq M. King, "A Self-Testing Approach to Autonomic Software" (PhD diss., Florida International University, 2009).

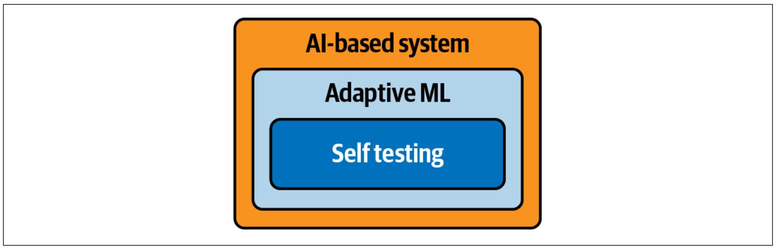


Figure 5-3. In the long term, self-testing is an implicit feature of adaptive AI/ML systems

Conclusion

Software test automation has a bright future ahead of it, and I believe it's largely due to AI. Throughout this report, you've seen several examples where AI is already helping to bridge the gap between human-present and machine-driven testing. AI is testing at different technical levels, validating quality attributes such as performance and accessibility, and allowing automated testing to reach areas like gaming, where it has been lacking for some time. These advances are not only important to the software-testing field but may come full circle to benefit AI itself. AI-based systems that incorporate online and adaptive learning will need an automation framework that is dynamically adaptive and embedded in the runtime environment. The only lingering question that I still have about that future is, who will test the AI that is testing the AI? To avoid this infinite loop, I hope that the engineering community continues to make investments at the intersection of AI and software testing, including the role that humans will play in a future where AI tests software.

About the Author

Tariq King enjoys breaking things and is currently the chief scientist at test.ai, where he leads research and development of the core platform for AI-driven testing. Tariq started his career in academia as a tenure-track professor and later transitioned into the software industry. Tariq previously held positions as a test architect, manager, director, and head of quality. He has published over 40 articles in peer-reviewed software-testing books, journals, and conference and workshop proceedings. He is a member of the ACM and IEEE Computer Society, and he serves as a board member and keynote speaker for international software engineering and testing conferences.