test.ai

The QA Director's Guide to TestOps

Jason Arbon
CTO and Founder, test.ai
February 1, 2021

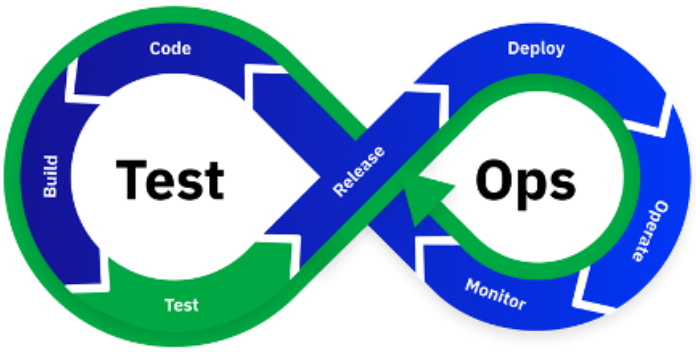
Notice

Test.ai reports, white papers, and legal updates are made available for educational purposes only. It is our intent to provide general information only. Although the information in our reports, white papers, and updates is intended to be current and accurate, the information presented herein may not reflect the most current developments or research.

Please note that these materials may be changed, improved, or updated without notice. Test.ai is not responsible for any errors or omissions in the content of this report or for damages arising from the use of this report under any circumstances.

Table of Contents

Introduction — The Imperative for TestOps



The development and operations (DevOps) model is now the mainstream for technology companies due to its speed and efficiency. The legacy waterfall has been replaced by Agile® software development methodology, the close cooperation of cross-functional teams, and true integration with operations.

Software best practice is no longer concerned with the quarterly or bi-annual shipment of monolithic code releases. Instead, new functionality is continuously delivered to customers through software as a service (SaaS) clouds and weekly downloads of new software versions in the app store.

The user experience remains of critical importance. Is the user able to accomplish their goals? Is the product working correctly? Is it satisfying to use? These types of quality goals can only be represented by end-to-end testing. Unit testing by the development team has improved substantially, yet measuring the quality of the user experience has fallen further behind.

Manual testing is costly to implement, expensive to scale, and produces imprecise results. In many cases, such as in support of app store testing, gaming, and many other applications, it is unable to scale with the volume or, critically, speed of modern DevOps. Manual testing takes us full circle back to the reason automation is so critical. Manual testers cannot scale to meet the needs of an app store, major gaming provider, or global enterprise.

Legacy software test automation processes and tools have been unable to maintain pace with the speed and increased efficiency of modern DevOps. In fact, the gap between the two is increasing. Quality assurance and engineering teams do not have the time to validate the quality of the end-to-end user experience. The windows of time available for end-to-end testing are weeks, even months, longer than the required software release dates.

Software testing tools are complex, slow, and quick to break. Legacy technology like Selenium and Appium has not changed over the past decade. When you need them most to test a new release, the hard-coded automated scripts will often fail and require diagnosis and update. The result is that the software is released and the end-to-end experience testing is done by the consumer. How can placing the burden for end-to-end experience testing on the consumer produce a better result? How will this impact the consumer's brand experience and, ultimately, revenue?

This paper will review the challenges and introduce the key principles that define the test operations (TestOps) imperative. TestOps will enable your organization to accelerate testing to match the speed and pace of modern DevOps. The resulting benefits from moving to TestOps include competitive advantage, improved efficiency, higher quality, improved customer experience and satisfaction, and increased value for your brand.

This paper will also introduce the use of artificial intelligence (AI) as an important enabler for the TestOps vision. AI provides a new generation of tools with the resiliency, speed, and scale that enables end-to-end experience testing prior to the release of the software.

Formalizing TestOps

There is a growing gap between the legacy processes to measure and improve software quality assurance and modern DevOps. The quality increase promised by DevOps has proven difficult to deliver.

The industry must formalize what TestOps means. TestOps is a new term backed by compelling need and empowered by a new generation of software testing tools driven by AI. We've seen development and operations merging together and moving in tandem. Yet, at the same time, the testing methodologies have been left behind, a secondary background process that cannot keep up with modern DevOps. TestOps presents a more modern way to deal with the increased cadence of software development and the weekly shipment of new software releases.

The best solution for us is to operationalize software testing. We need to move testing forward to a point where, once again, testing is a relevant part of the development cycle. Let's talk about TestOps, our vision for what it is and where it can go, and, most importantly, the problems that TestOps can solve.

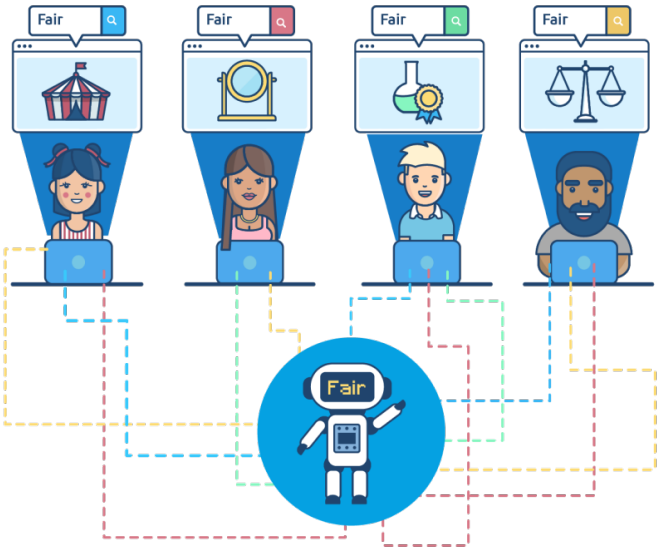
Tests and tools produce inconsistent results

Tests and the legacy tools that go with them produce inconsistent results and fail for all sorts of reasons. People frequently try to rerun them, often without success. Small changes in the test case almost immediately render that carefully developed test inoperable. In order to help mitigate this problem, the legacy test automation suite vendors work endlessly to build better frameworks. They have invested hundreds of millions of dollars to build these new and improved frameworks. Yet most legacy testing tools are still inconsistent in their performance. As most of the quality assurance testers and software development engineers know, when a vendor or a peer tells you that their tests are perfect and run all of the time, they are exaggerating. There are few test automation scripts that reliably make it more than a couple of weeks without having problems. This is especially true if the software being tested is updated every week or two.

Testing is a slower process than it needs to be

Testing is a slower process than it needs to be. Why? Because most of your energy is invested in manual processes. If you write a lot of automated test cases, you have a considerable amount of wait time with which to contend. Many testers spend time clicking (and clicking) and waiting for the app to catch up to these manual activities. You usually cannot get these events to run in parallel. When you can, the costs increase. These tests fail often and still need to be debugged yet again before you can get the test final results. On a good day, testing is slow compared to the cadence of modern DevOps. On a bad day, it is almost frozen in time.

In the old days, 15 or 20 years ago, the waterfall model had a clear plan for the testing processes. The waterfall model dealt with monolithic code produced in a discrete release. Maintenance releases would come out periodically. There was not much in the way of the cloud that was just emerging with software as a service (SaaS) offerings. You had unit testing in engineering, integration testing to see if the individual pieces hung together when assembled, and then quality assurance testing to see how the app ran end-to-end with sample data entry and sample databases. Testing followed at the end of the development cycle. It was part of the plan prior to shipment. Everything had to stop to complete testing and recursively validate the correction of the software problem reports. The test automation suite tools available at that time worked reasonably well with that cycle. Once the quality reached sufficient levels according to some set of metrics, the engineering team would declare the software ready for release.



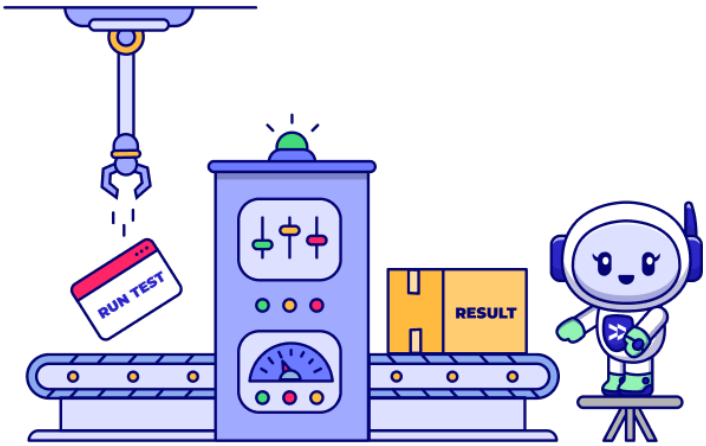
Today, those same test automation tools used over a decade ago are still attempting to follow in the wake of modern DevOps, like Jurassic Park dinosaurs wandering down a modern highway.

Yet Agile has long gone mainstream. Continuous integration and continuous deployment (CI/CD) processes dominate the day. CI/CD bridges the gaps between development and operation activities. Automation is wrapped around the building and deployment of applications. The cloud is here. New applications are getting updated daily and legacy approaches to testing have been unable to keep up. In fact, our customers are doing the testing for us, and our quality standards have gone out the window.

Testing tools are complex

Testing tools are complex. Most have more dependencies than any tester can remember. Truth be told, they are too complex for the modern software engineering world. DevOps has simplified and streamlined, and legacy testing has not.

Complexity is also driven by all of the dependencies in these tests. You have dependencies on the product code, third-party libraries, and third-party services. There is too much going on with all of these dependencies; the result is instability.



Automated tests break when you need them the most

Applications are constantly changing. A button moves or the flow changes; perhaps a new feature is added. That is the most likely time during which your existing test cases will break because they are hard-coded to the application implementation. It breaks at the exact moment the development team has changed the application and asks, of course, does the test work? Can you test this? And the answer is “please wait, I have to debug my tests.” It is exactly at that defining moment of truth that the test won’t work. These hard-coded test cases are complex and require time to correct.

Adding insult to injury, a developer, product manager, or manual tester has just tested the functionality manually and says, yes, it works. It is just plain embarrassing. This happens all of the time to test automation engineers. It is precisely when you need them most that automated tests break. This cycle repeats itself over and over.

Products release before testing finishes

The team at test.ai have worked with many of the largest, most sophisticated, and richest companies on the planet. The biggest issues are the same. They ship software before they finish testing it. Most push out the code every two weeks. A test cycle, even an abbreviated one, may take several weeks to execute. Bugs cannot be identified quickly enough to be fixed in each ship cycle, and, worse, you are finding them after the customers find them.

How does this look to executive management, product management, and the line of business? These people are getting frustrated because development and operations have become incredibly efficient with quick shipment and go-live for code, yet testing falls further and further behind and has become, in the most extreme view, hardly relevant to shipping software at all.

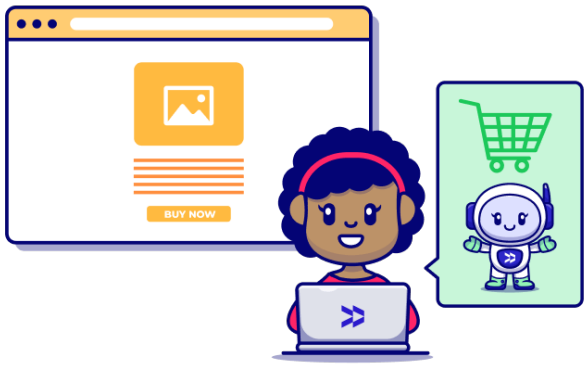
End-to-end User Experience is What Matters Most

Testers and quality assurance teams need to truly represent the end user and the consumer. They need to make sure it works for and is easily understandable by the user. They need to empathize with the end-user point of view.

Today, we are losing this battle. Most of our effort goes toward making sure the applications work like they did yesterday. There is little to no time to sit back and be empathetic. It is all we can do to execute the same tests over and over again manually, and now and again have test automation that will work when you need it.

The DevOps revolution has brought big changes for developers. There are more application program interface (API) tests and unit tests being written by skilled developers. They happen in real time with the code design and implementation. The beautiful part is that these tests are easily automated, robust, reliable, and run faster and in concert with the continuous integration environment used for build and deployment. They're elegant, they're fast, and they're easy to debug.

But none of this matters if the end-to-end application doesn't work well for the user. One only has to look around to see that most software is lacking. Testing teams are not keeping up with developers prior to the software release. We know from life experience that, in the modern software testing world, API and unit testing alone are not enough.



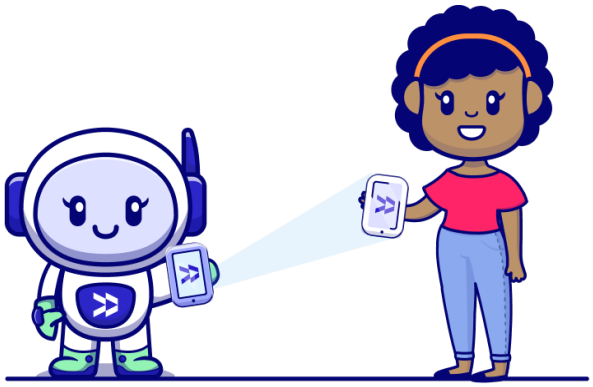
Examples of this are abundant. One of our team members recently went to buy a new car. To him, the coolest feature of the car was the connectivity. He wanted to use the mobile app to unlock the car and use the cameras. Unfortunately, the app did not work, and the user experience was awful. I'm sure the car company's APIs are great. I'm sure its operations team believes the feature works and that it has stacks of unit tests for shipped cars that prove it. But, in the final analysis, the end-to-end experience doesn't hang together for the user. The app is a bust.

End-to-end testing is far more than just a bunch of API and unit tests. API and unit tests are an important step on the way to delivering the best user experience, but, in and of themselves, they fall far short. We need complete end-to-end testing. The end-to-end experience for the user is key.

Another example is a major travel website. It has a million things that need to work to deliver a successful booking and a satisfied customer. It has a mobile app that's critical. The company is swimming in oceans of regression tests for various portions of the app. It has all sorts of test cases. But, once again, none of that matters if you cannot reserve a car or book a hotel with the application. If the end user cannot find what they need or are unable to use it, it doesn't matter.

Gaming may be the tip of the whip for pushing the technology and forcing the issue. End-to-end testing with a game can be quite dynamic. We've seen many examples where the lack of end-to-end testing results in a poor user experience. There is a lot of unit and API testing to make sure game mechanics work correctly. The game probably tests out mechanically perfect at the API unit level. But these tests don't matter if the end user cannot play the game and if the game doesn't work correctly. The game engine needs to have the correct score, the correct number of objects on the map and in the treasure chest, and the right amount of ammo in your clip. When a user equips a new weapon, it needs to be the selected one. But is it displayed correctly? Does it look good? Is it fun? Can you move correctly in the environments?

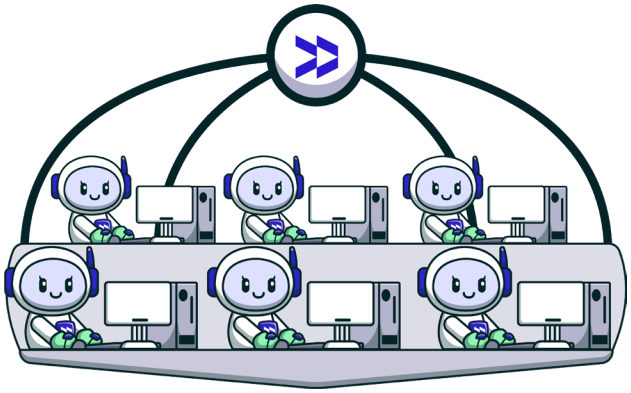
The animations must work. And the in-application purchases must work. There is nothing worse than buying an avatar or bling upgrade and finding that it does not work correctly. The customer doesn't know (or care) that the code was written two days ago and the manual testing has not found it yet. Customers will be unhappy and want refunds because the brand experience was poor. The end users are not paying for an internal API or gaming engine. They are paying for the end-to-end experience. That is where testing and quality assurance need to focus.



Now we are getting closer to the crux of the issue. These are the sorts of things that need to be tested end-to-end. End-to-end testing is difficult because it relies on the entire system being up and running when you need it, not when you fix it. Everything must be implemented, coordinated, and communicating to the peak of perfection.

The terrible truth is that most companies don't have any real end-to-end tests, and those that do only have a few because they are so difficult to design and so expensive to write and maintain. And they are still not running when you need them most — they remain in the distant shadow of the DevOps process.

Passing over failing



You need to think outside the box. Traditional testing is about finding bugs. This is no longer the key metric. In the TestOps world, what you want is for everything to be passing your tests every time. You want the product to work. You want it to be great, you want it to be reliable, and you want it to be able to ship. Rather than writing a bunch of complex test code that tries to put the system into an unnatural state where it will fail, it is more important to make sure that basic features and key functionality continue to work every time a developer makes additional changes. TestOps tools need to work on the developer's machine, the integration build, the production monitors, and end-to-end testing.

Quick over exhaustive

Tests need to execute quickly. A quick and sufficient determination of quality is more important than finding an exhaustive measure of quality. It is far better to have 10 key measures of quality in your top 10 test cases that are so reliable that you can run them in any environment in every new build and know that major functionality still works. This is in sharp contrast to a suite of 1000 test cases many of which never pass and all of which require continuous maintenance. Percent pass as a metric can take your eye off the prize. Ninety percent pass versus 95 percent pass doesn't tell you what you need to know. The real issue is whether or not the open issues are forcing a delay in shipment. We want to know about these issues.

Test execution can waste valuable time. During that time window, the developer may lose focus and become bored waiting for the test results. Everything can add latency and delay, so the shortest possible tests are our ideal goal. This also makes them easier to debug and more easily understood.

When you focus on basics, your developers can quickly see if a new build is broken. The alternative is to wait for a bunch of long-running tests with dozens of steps, some of which will likely break. The focus in TestOps is, thus, on the simple things that yield quick and important results.

Easiest and most reliable tests

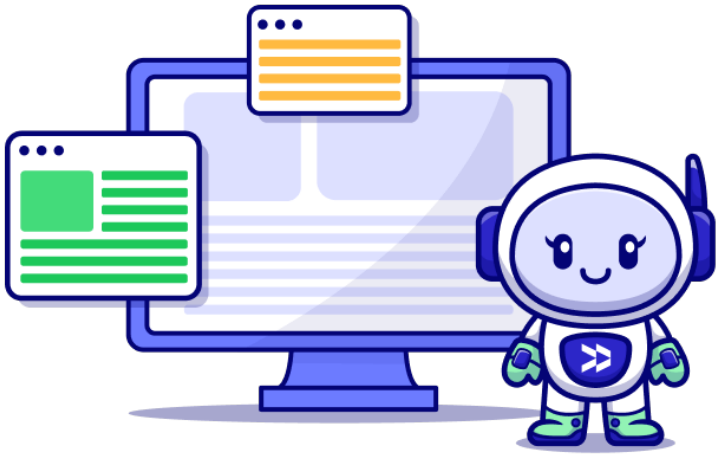
Only the most reliable and easiest tests to write and maintain should exist in the TestOps world. They should be executed often because the more often you execute them, the sooner you find the bugs or regressions and the faster you can fix them. The sooner you know that the build works, the faster you can ship and release it or add new features. Executing key test cases often is critical.

You may have several things to test with one of them obviously the most important, such as, for example, signing into the app. Sometimes sign-in can block the testing of other areas if the sign-in process cannot be satisfactorily automated. In such a case, test teams should instead prioritize what can be more easily automated and build those reliable simple tests. It may be better use of your time to write tests for search functionality, for launching the device matrix, for performance, for finding products, etc.

Log-in is essential and business-mission-critical, but that doesn't necessarily mean that it should be part of TestOps. In the final analysis, that might be best left for humans to do manually. Or perhaps you leave it in the background with test automation flows that look for corner cases and sign-up. Just because it is important and mission-critical does not mean it should be in the operational part of testing and integrated with DevOps procedures.

Less is more

You can get a good measure of quality with just a small number of test cases. And that means less time required to execute debugging, less maintenance, and better return on investment for time spent. While testers are used to thinking in terms of adding to the portfolio of tests, in the faster-paced TestOps world, we want to keep the number of test cases small.



Simple over complex

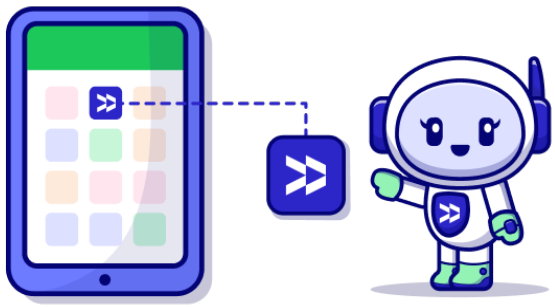
Testers often like to add libraries and other integrations. You want to avoid these types of dependencies to make your tests more reliable and easy to use. The use of too many application program interface (API) calls under the hood, calls to libraries, or connections to some cloud service will negatively impact the speed and reliability of test cases and the debug capability.

Simplicity is the goal, as complexity always reduces reliability. If you know where you want to click on the xy-plane in an application and you also know that this is unlikely to change, it is better to just send the click to that xy coordinate. Instead, often teams will load up a huge legacy framework like Selenium or Appium. These tools are full of complexity with messaging, queuing remoting calls, and more. Avoid these complex dependencies wherever possible.

Avoid frameworks

The more frameworks you have, the more compatibility and interaction problems. You're going to have more complexity in terms of debugging, understanding what happened, and determining what went wrong. Use the most primitive calls to drive the application or measure the application state as possible.

Frameworks make things less reliable and more complicated. They mean fewer people are able to debug that code and understand how it works. The best practice is to avoid complex frameworks for TestOps.



Avoid custom testability

Also, avoid custom testability — don't load up your quality assurance team with development tasks. This adds considerable work. We have to add these extra attributes, schedule and attend meetings to coordinate with the development team, and then document all of it. Every time you get a new feature, you have to remember to add this extra task. This slows down development and, most of the time, is unnecessary.

Alerting instead of reporting

The instant a test case starts to pass or fail, or the performance of some scenario changes, you need to alert somebody immediately. You don't want to just post it to some server or log somewhere. You want to make sure it's rapidly actionable because ship cycles are measured in days. You need to get this information to the right people to reverse the change or to make a new code change to fix the issue as soon as possible.

Nobody wants to deal with test reporting dashboards, with the rows and rows of red and yellow and green. This is not what people want to want to look at.

When you find out that a particular check-in caused a regression and you know that there's an area owner for a particular feature in the application, you send them a text message, an email, or even a mobile notification in their own app. There are real-time ways to communicate failures with people and alert the exact people that need to know. Don't just put it on a dashboard and hope that somebody refreshes and triages it. Get directly to the people.

See Children	Build Number Package *-Class-TestMethod* Names	16	15	14	13	12	11	10	9	8	7	6	5
⊖	org.common.samplea	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	N/A
⊖	SampleATest	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	N/A
	testA	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	N/A
	testB	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	N/A
	testC	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	N/A
	testD	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	N/A
⊕	org.common.sampleb	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	N/A
⊖	org.common.samplec	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	N/A
⊖	SampleDTest	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	N/A
	testA	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	N/A
	testB	Skipped	Skipped	Skipped	Skipped	Skipped	Skipped	Skipped	Skipped	Skipped	Skipped	Skipped	N/A
	testC	Skipped	Skipped	Skipped	Skipped	Skipped	Skipped	Skipped	Skipped	Skipped	Skipped	Skipped	N/A
	testD	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	Passed	N/A

Testers and Developers — Different Missions, Similar Goals

One of the new mantras, resonant with management, is “we are all going to be testers.” But who is really going to write these test cases and implement TestOps? It has to be the quality assurance and test team. We know how to read a test case. We have empathy for the end user and the end-to-end experience. We know the end scenarios for the application, and we have the skills to keep the code focused and simple so it is reliable. Testers are the best people to be building these TestOps automation regression suites and end-to-end testing.

Software developers and engineers have refined technical skills. They think about the classes in the code, the objects, and the methods they will use. They must have a deep and detailed view of the application components they support. When it comes to the unit and API testing, they have by far the strongest skills and the most relevant knowledge and experience.

Execute often

You should be running tests with every new software build; you want to find out if there’ve been regressions and then find these as soon as possible. The faster you find bugs, the faster you can fix them.

You want your tests to run on every build and even on the dev machines. When developers are running a local build, it’s great if they can run some basic verification tests. The developers can kick off before check-in, perhaps run a test or two end-to-end locally.

Testing labs must be fully available and complete

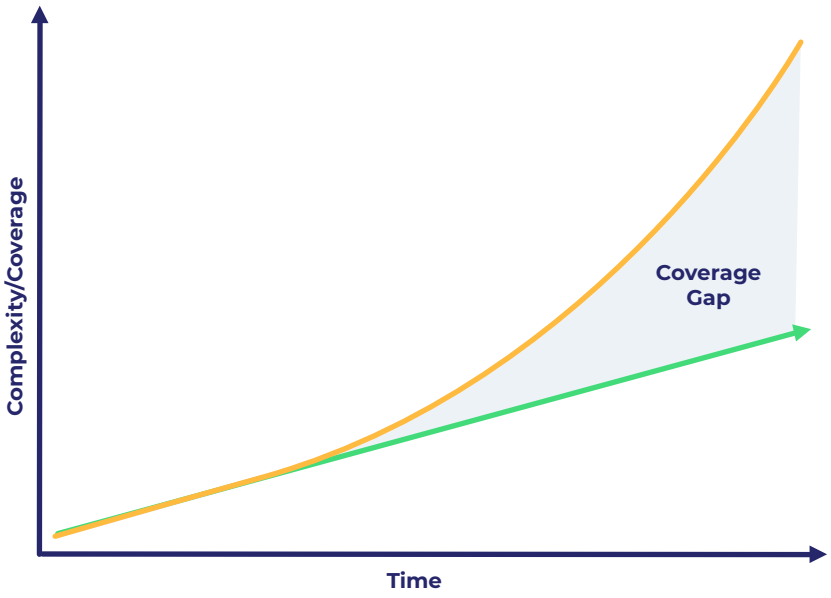
It is far more expensive to interrupt testers and developers than it is to invest in a complete and capable testing lab environment. This is not just about hourly rates. It is about the speed of development and making overall goals. Rather than have a collection of devices hanging off a single controller machine, you go one-to-one. For every iPhone, you have a controller like a Mac. It may seem expensive, but it’s far cheaper than engineering time. This strategy nets real return on investment. It speeds up your engineering effort because there’s no highly limited contention for resources on disk network or CPU.

Don’t even consider being cheap with your testing lab at scale. Think about building a small lab that’s hyper-reliable and overbuilt to a degree. It will save you engineering time, money, budget, and frustration. Your team could try to rely on a third-party infrastructure, but this introduces an entire set of new problems, unknown variables, and additional risk. TestOps best practices don’t want to see repeat failures. You will be running tests so often that, if there is a certain failure rate, you will see it. Failures caused by limitations in the testing lab will be there continually.

Leverage Artificial Intelligence (AI)

AI provides flexibility and resiliency while reducing instability to allow tests to run continuously across platforms and devices. The principles of TestOps require us to think differently about how we test software. Testers can no longer afford to spend their time maintaining tests that break randomly or only kick off tests every once in a while. Artificial intelligence test automation tools can be more robust and resilient to expected changes and can run automatically with each build.

AI is a sharp contrast to the hard-coded scripts you are likely using in Selenium or Appium. AI can recognize elements on a screen just as a human tester would and see the buttons, images, and text. The AI is then taught what to do with the elements even if they move locations or even design, in contrast to hard-coded scripts that break with each change.



Testing needs AI

Features
Complexity increases exponentially as new features and states interact with existing features

Tests
Test coverage grows linearly because tests can only be added one at at time

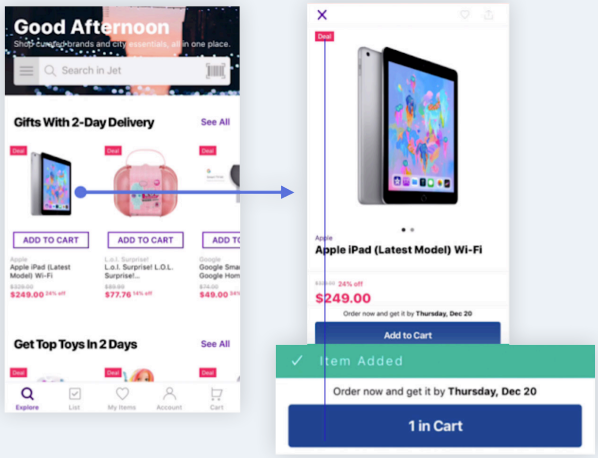
Growing Platforms and Devices
It is not getting any better or less complex

Write once — use many

Tests that you build with AI are reusable across platforms and different runtime environments. You build it once and gain high resiliency to UI changes. You can take a test you have developed with AI for iOS and, in turn, deploy it to Android. AI is not just a nicer way to implement testing. It is critical to get to the flexibility, scalability, and reliability levels that you need for TestOps best practice deployment.

Element Recognition

ML trained to recognize elements just like a human.



User Flow Execution

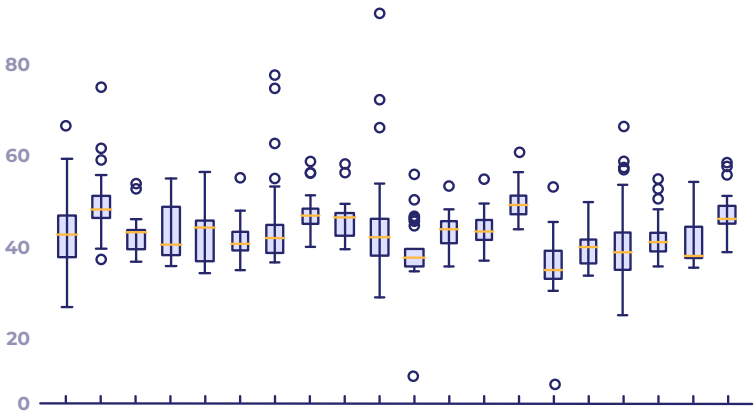
Test actual tasks. Automatically works across hundreds of apps.

TestOps requires the reliability of AI

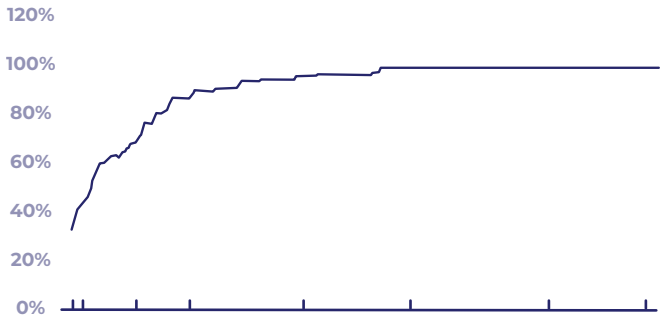
A core principle of TestOps is consistency in test results. Old methods of automation simply won't stand the test of time because they provide too many false positives and negatives. DevOps and TestOps don't have room for downtime and maintenance of unreliable tests. AI is necessary for TestOps resiliency and consistency.

We need to get end-to-end tests running on every build in the normal operations and development flow. Old methods won't work, but people may try them anyway because they're intimidated by AI and ML. Selenium and Appium are the old generation of legacy hard-coded testing tools. You move step by step with each line of code. These legacy tools are not adaptable, unlike the massive power of AI and machine learning.

The TestOps tester will be required to learn about AI and how it applies to testing. You don't need to be an expert in AI or ML and you don't need to invent it. Take the time to learn the basics of AI and then use it to build your tests. When you learn about AI and the tools that support it, you will run alongside developers at DevOps speed.



Average Time to Load for Data Centers on September 13



Successful Game Launch by Game Name

AI for performance and scale

Traditionally, UI-based testing has been deemed too unreliable to be used for performance and scale testing, but the reality is that this is how users will experience apps. The alternative testing method has been to use API testing, the idea being that API tests are much more reliable and easy to automate. The challenge is that our users don't experience our products through the APIs, they experience them through a browser or an app. Without the power and resiliency of AI, we are stuck hoping it will all come together for the end user. With the power of AI, we can run tests at massive scale and gather performance metrics to relay our user's experience to our dev teams.

Here's a view of some of what we do during testing. Let's take a look at a single application across multiple data centers. Along the x-axis in the above left figure, we see different data centers. We have cropped the graph to protect the innocent. It is important for you to see these latency numbers for the same application on different data centers. This is the end users' experience of latency. Not a single API. A single action from a user may require four or five different APIs to be called, some in parallel. And this is the net experience of the user, regardless of whatever you think. You may believe your latencies are fine; they are often not when viewed from the end-user perspective.

The figure to the right shows another metric we tested from the end-user perspective that includes the launch of games in the cloud. The x-axis shows different games. The y-axis shows how often that game launches successfully. You are seeing TestOps tools quantifying everything. We can quantify quality, performance, latency, and crash rates.

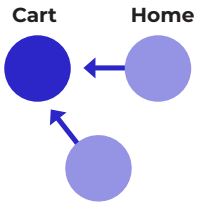
AI enables you to launch hundreds of different applications and measure across different data centers, including their launch success rate. This data tells you which apps to focus on, which have the worst performance metrics, and which perform fine. You can reliably rerun this day after day, make improvements, and see if your infrastructure is getting better or worse.

Using TestOps, you're testing the end-to-end user experience. You know with certainty what your users are actually experiencing.

Test.ai Transforms Testing

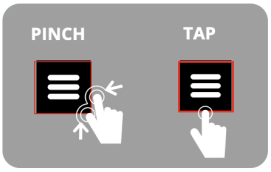
Model

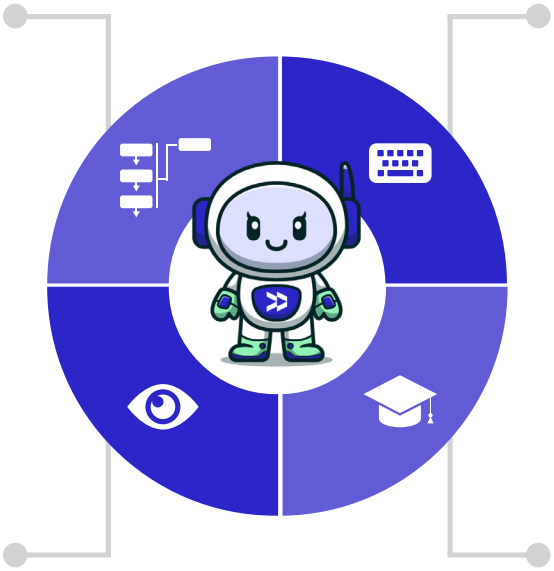
Keep track of the application and environment state to help deal with uncertainty and partial observability.



Test

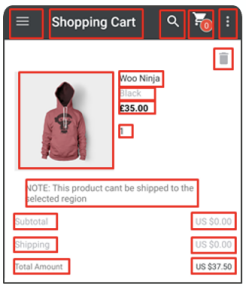
Generate actions to navigate the application UI, input data, and verify observable outcomes.





Perceive

Perceive the environment including the application UI structure, behavior and any differences over builds.



Learning

Direct feedback in the quality of an agents actions to help improve future actions.

Punish Reward

Rewrite Reinforce

Test.ai is transformative — the gap between the capabilities of legacy test automation tools and Agile® development practices typical of modern DevOps is now eliminated. Applications and new features can be brought into production confidently, knowing that all new capabilities have been tested before shipment. Customer satisfaction, brand reputation, and revenue can all improve as a result.

Test case creation

No coding is required at all. Test.ai provides a simple drag-and-drop codeless user interface for labeling items and creating test cases. Machine-learning-based element detection and classification eliminates the need for CSS, XPath, and IDs to find elements. Test.ai detects elements visually — just like a human. Our AI-bots are trained with reinforcement learning so that they can intelligently navigate without specifically defined steps. The test creator also supports codeless, behavioral-driven, test case definitions.

In rare instances where it is required, test.ai also allows custom Python scripts for additional control over logic in test case steps. The Abstract Intent Test Syntax is a publicly defined test case format that allows for the import and export of test cases. Preconfigured Smart Automated Tests provide a base set of tests from which additional AI-based exploratory or cross-app and platform tests can be run.

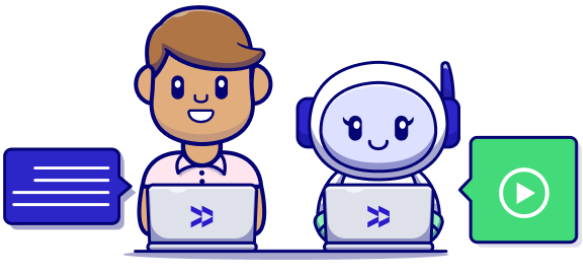
Visual reporting

Screenshots are taken at each step, and each interaction is highlighted on the screenshot giving a visual user flow result for each test case run. Test.ai provides test flow results for each test case run. This includes performance and device metrics, including screen load times, CPU, and memory usage. Test.ai also includes an API for integrating custom reporting.

Test.ai's video game ML object diffing allows you to test thousands of assets automatically. Video games have thousands of assets, and validating each one is impossible manually, but thanks to our ML Object Diffing, we can test video game assets to make sure they appear as desired based on the original creations.

Scaled test execution

Test.ai scales to support thousands of VMs and applications in a single run. Full analytics and reporting are provided on each app, device, and test case.



Human Interface

The Human Interface provides a unified GUI that allows engineers to design and develop automated tests for a software application. It includes subcomponents for training the AI-bots via labeled data, authoring and executing test cases, and viewing and analyzing the results. This layer provides a view of all unique screens and elements discovered by the AI-bots and allows users to define their own data labels for UI elements or apply bot-generated label suggestions. Test specifications utilize these labels for ML-based, UI element selection during test execution. The execution interface provides settings for configuring the AI-bot's test suite, environment, and execution mode.

Application interaction

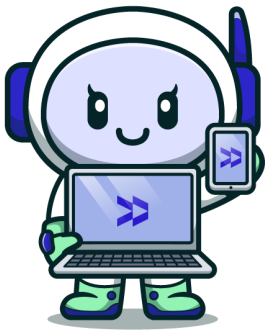
Application interaction is responsible for coordinating autonomous AI-bot exploration and test execution. Test.ai supports two exploration modes for the application under test: fully autonomous and human-interactive.

- Autonomous exploration involves one or more AI-bots automatically crawling the application under test and collecting screen and element information for future labeling.
- Interactive exploration keeps a human user in the loop, allowing the user to decide which parts of the application under test will be explored and in what order.

Supported Devices and Platforms

Test.ai can support goal-based or scripted execution on virtually any device where an image or video capture is available. We have developed and formalized support for automated testing of an industry-standard list of platforms and software:

Device Category	Device Type
Mobile	Android®, iOS®
Simulated and Real Device Support	Android® emulators and iOS simulators and any real devices, Apple® TV, Apple Watch; integration with cloud device labs
Gaming Platforms	XBOX®, Playstation®, Nintendo® Switch
Browsers	Chrome®, Safari®, Firefox®, Windows Edge®



test.ai

About test.ai

Test.ai is a leader in building AI-powered software test automation tools that help testers, developers, and business stakeholders accelerate the release of high-quality applications. Test.ai replaces legacy test automation tools that don't work well, fail often, and are hard to use. Our AI-powered bots build the tests, scale them from one platform to many, and maintain them as your applications change. Artificial intelligence enables our bots to do all of the work — no scripting or coding is required. Our customers include some of the world's largest technology companies, major app developers, app stores, and some of the world's largest app platforms manufacturers.

About Jason Arbon

Jason Arbon is the founder and chief technology officer of test.ai. Jason has worked in development, test, and product management roles. Jason has tested browsers, operating systems, search engines and apps at Microsoft, Google, and uTest. Jason has always been obsessed with software testing at scale and is instilling that passion in AI-bots today.