USE CASE: MACHINE CREATED AUTONOMOUS TESTING

by Kevin Surace, Appvance

Key statement: Quality engineering is not about creating and managing an ever-increasing number of scripts, but to simply detect more defects with less human labor. Using a huge automotive ecommerce website as concrete proof point, we demonstrate how AI can provide ten times the application coverage of typical automated findings. It's as if an additional 100 team members, all of whom are excellent bug hunters, are scouring the UX and API calls as well as validated outcomes for any flaws or unexpected results.

The dream of fully autonomous testing was indeed a crazy idea in 2012, until Appvance launched the world's first fully autonomous testing in 2018. What is autonomous testing? It isn't about a QA engineer or developer writing scripts and then kicking off with Continuous Integration (CI). That's a concept we have had much of since 1995. It's a generation, arguably a magnitude leap, beyond writing and maintaining scripts.

The driver to autonomous testing

The reason for this requirement is quite simple. And follows this math:
Assume a waterfall process with eight weeks or 320 hours to update, maintain, and execute tests that were possibly developed several years ago. Approximately 98% of all testing is now performed on existing apps that are being updated. The automated testing we perform on a consistent basis throughout the industry averages approximately 10% application coverage. This entails assessing each user activity, page, and unique state in nearly any flow.

Why don't we have more application coverage? Because it requires substantially more effort (around 10X) to maintain that many scripts. So, we settle for low application coverage but high test coverage. Test coverage was defined by an apparently magical Business Analyst who knows what every user does. BUT we know this to not be true. How? When users find issues those are almost always issues in using the application in ways we didn't test. Because had we tested it the way they use it we would have seen the same functional bug. If we had limitless resources, we'd like to increase application coverage to 100%. (or 10 times what the industry average is today).

At the same time, management is driving to Agile and DevOps (and BizOps) processes. Reducing testing time from 320 hours to one hour demands a 320 times increase in team

efficiency. And if we eventually aim for 100% application coverage (which means that almost no defect will escape our tests), that's another tenfold gain in productivity. That's a 320 x 10 or 3200 times gain in productivity if we aim to detect all issues in one hour. Whether your organization is moving there slowly or quickly, you are heading in this direction.

In 2012, the math indicated that this would already become the big issue, and no one would hire 3,200 additional quality assurance personnel to augment the existing workforce (of course). And 3,200 workers would be incapable of coordinating to produce a 3,200 times increase in productivity in any case.

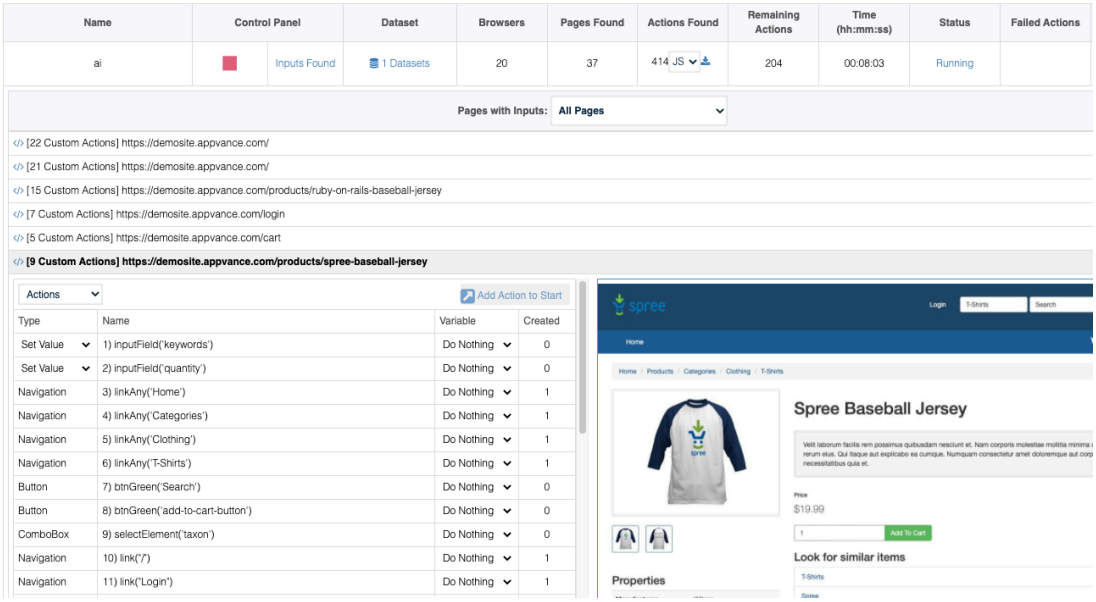
What I've observed is that teams begin to reduce the number of tests they conduct in order to reach the 1-hour deadline and hope they checked the right things. And as a result, users frequently find more issues than ever before and revert in production.

Our approach to providing Augmented Intelligence

This here is where artificial intelligence excels. While the term AI (artificial intelligence) is frequently misused in marketing, the notion refers to a wickedly fast and intelligent machine capable of performing numerous jobs orders of magnitude faster than humans. I prefer to refer to AI as truly AUGMENTED intelligence. It augments us humans so that we can concentrate on being its overlord and studying the outcomes, rather than struggling for a day with a selenium script. Allow the machine to perform the more or less monotonous tasks.

We addressed the development of an AI capable of independently creating tests and detecting defects as a 3-part challenge.

1) Train the AI on your application one time (perhaps production release) to establish a baseline of UI and API requests/responses
2) Run a new AI Blueprint at each new build, allowing the AI to learn what changed on its own and create scripts based on models and limits provided in training
3) Fine-tune the scripts based on learning from standard API production logs and upscaling those API requests to UX actions.



Example of an AI Training Session

We discovered that we needed to replicate human-like learning, behaviors, and decisions in order to develop end-to-end testing that did not require human intervention. We identified 19 potential application areas for machine learning. As explained in the "*Demystifying Machine Learning for QE*" chapter, this is referred to as machine learning, as opposed to deep learning or neural nets, which require massive clean data sets. Your application is not a good fit for the neural network model. We outline all these processes in 3 seminal US patents: 10204035, 10552299 and 10628630.

This technology, known as Appvance IQ (AIQ), is used across all types of applications, ranging from consumer-facing to SAP, Salesforce, and ServiceNow.

AIQ architecture

AIQ is a highly scalable system consisting of a controller (which test users access) which launches unlimited testnodes. These testnodes actually execute the virtual users. AIQ can be deployed in the Appvance cloud or behind the client's firewall. The AIQ AI library has already learned how to interact with thousands of UI actions from prior encounters as well as human assisted learning. These libraries get updated for all users every few weeks. The critical point is that the additional learning is focused within each build of an application. It determines what has changed and verifies that the change is legitimate. In some cases, the system only notes specific changes rather than decides if they are indeed bugs. In other cases, they are obvious bugs and tickets are automatically filed in Jira (or equivalent). Autonomous testing is kicked off at each new build by a CI tool (such as Jenkins) to support an agile or DevOps pipeline. And tests can be targeted as functional, compatibility, performance/load, or security tests, all withing the same infrastructure and execution engine.

A broad set of machine learning algorithms are employed to learn from prior builds and make decisions on current test creation. For example, a predator/prey algorithm is employed in creating UI scripts from standard server API logs. As on any application page, a user could take many routes. The logs reveal certain API requests which must be grouped into known groupings for all requests on one page. Then that logged grouping is scored as to its match to all actions a user COULD take. Because this is a nearly unlimited possibility, we continually score the top three "winners" and eliminate all others, then add additional possibilities and repeat the process until there is a single winner. In this way, we avoid overburdening a system with limitless alternatives and instead maintain only the top three, therefore constricting the problem. The algorithm is extremely quick, and it can resolve a single user action from hundreds of possible actions in less than a second. Bear in mind that machine learning is ultimately just math. We learn from the past, build a model and apply that to the new information, ultimately seeking the best match (or path or action etc.). We learned in the above example from both the current blueprint of the current build (the present) and possibly 50GB of API logs (the past). We must map the past to the present with the best match possible**. As a result, 5000 scripts are written in less than ten minutes, recreating the bell curve of real-world user flows from production and mapping them to the new build. Without any modifications to the application.**

From an architecture standpoint, a false positive should be impossible. Because it is actually a machine, the system will report only those bugs that you have permitted (such as validating an outcome which was not achieved, or an API response delay and many more). In practice, we encounter only absolute and intermittent bugs, not false positives. While certain bugs may be the result of the QA server's inability to handle a large number of threads, they are still bugs by any standard. However, the remedy would be to bolster the QA server in order to resolve them. However, the system will not report something that did not occur. It makes no judgments; it simply reports the facts.

AIQ is frequently used across the environment chain, including through staging and production. Additionally, it can be employed in the synthesis of synthetic APM.

Applying autonomous testing at a large ecommerce player

This case study is comparable to hundreds that have been published recently. This corporation handles several billion dollars annually of sales in the automotive space. They have a mature consumer facing website that contains catalog, search and purchasing functionality along with other information.

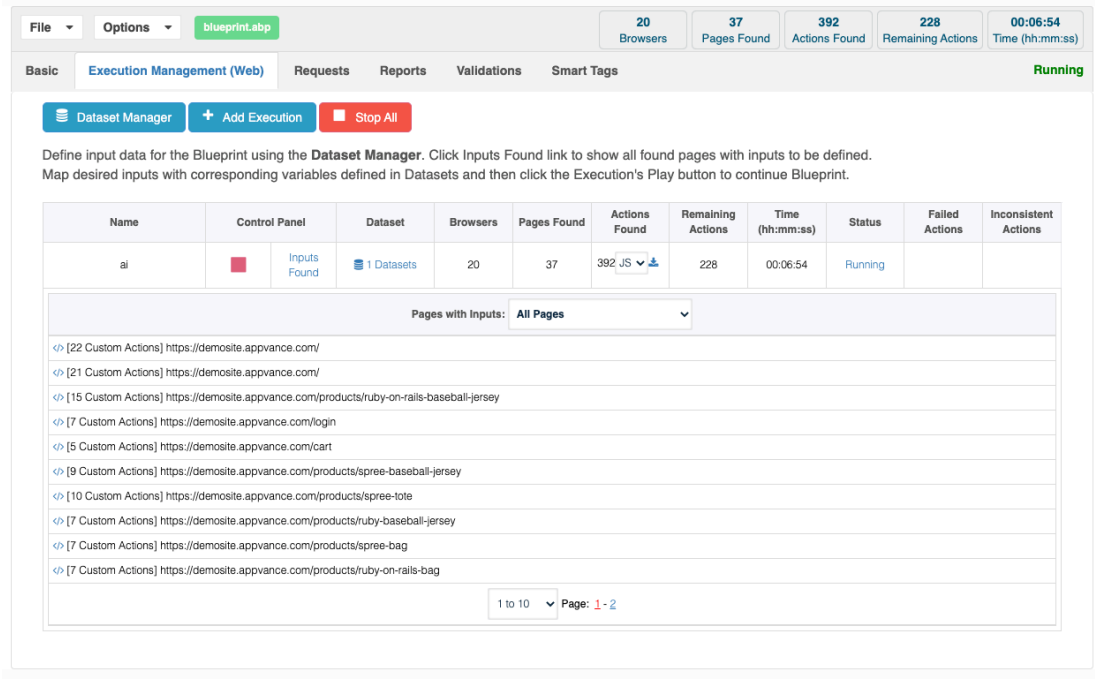
They have hundreds of automated selenium scripts as well as manual testers. They deployed AIQ’s autonomous testing to augment their existing test activities.

Setup (AI Training) for this web application took several hours. Training is accomplished by the autonomous engine walking through the application a few pages/states at a time and surfacing actions which users might take. These includes clicks, forms, dropdowns etc. As these are surfaced to the trainer in the AI workbench, the trainer can choose how the

system will handle each action. In most cases, we leave them to be acted upon as a human would. However, for certain actions, we may choose to limit interaction to never, or once per application or perhaps once per page. We mapped the fields to CSV data or to generated data. We did not want to test all 4 million inventoried items in this ecommerce application, so we specifically limited anything that appeared to be a new inventory possibility to only one time for the application. This allowed us to focus on the distinct pages and states, rather than repeatedly presenting the same inventory presentation page.

Additionally, AI Training considers validations, which refer to the fact that certain specific flows or input values which result in a particular outcome. This is a more abstract term than "asserts," as it applies throughout an application and through dozens, if not hundreds, of user flows. This is in direct opposition to the way we were taught to keep the number of user flows we test to a minimum and to test assertions only once. Here we have an extremely fast machine that is capable of writing and checking results 100,000 times faster than humans. Rather than reducing or limiting testing, we now want to prioritize maximal application and validation coverage.

When the baseline setup was complete, we were ready to execute a new AI blueprint on the next build. The blueprint was successfully kicked off and finished 42 minutes later finding 234 unique pages and hundreds of actions, writing, and executing several hundred scripts. The autonomous system follows a course as a human would. It accomplishes this by making educated guesses about every component, from interactions with thousands of elements to page state judgments to match scoring. Rather of relying on a single machine learning algorithm, we use 19 various machine learning algorithms, each of which attempts to solve small problems depending on past information about the application or applications in general. Bayesian curves, predator prey, recursive error reduction, decision trees and other traditional algorithms are employed. In this case, the system was asked to deploy up to 100 threads (bots) in parallel, each integrating with the other to not overlap user flows.



Example of a running AI Blueprint

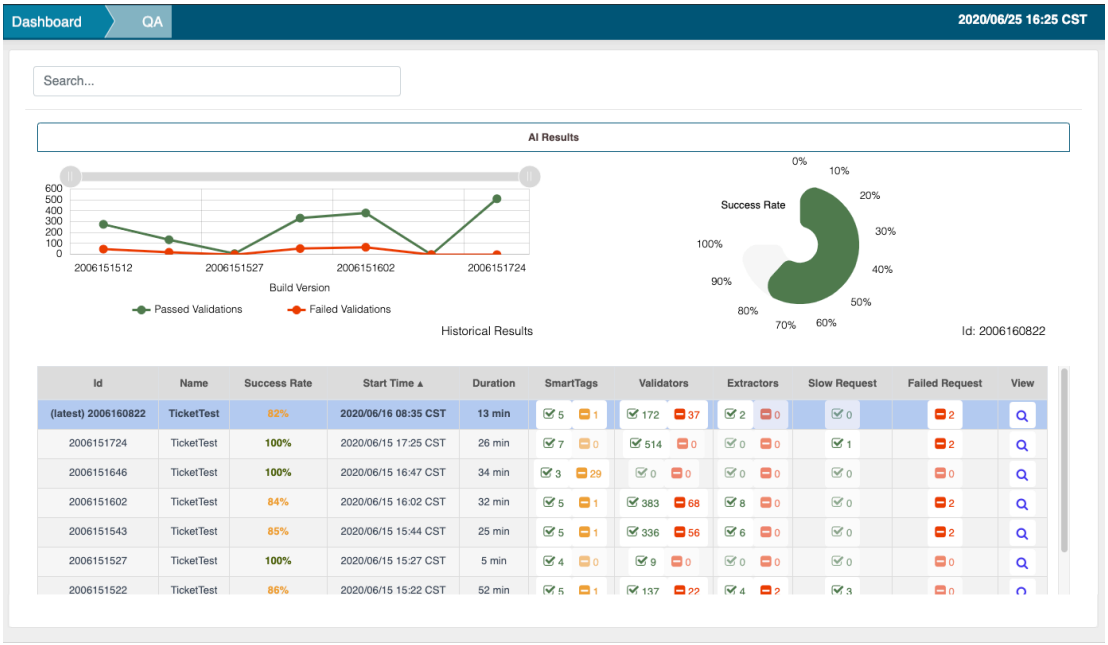
The results were in after the first blueprint was completed. 110 new bugs were identified. These UI and API bugs were not found in the teams traditional automated and manual testing. The bugs consisted of malformed API requests, API non-responses, 4xx and 5xx responses as well as some very slow responses. Each of these had a negative impact on the user experience in some way but had not been found through conventional techniques. The team's responses were typical:

"*These bugs aren't real* (it's a machine…it doesn't post false positives)."

"*These bugs didn't matter.*" Ah, but they did as you see.

"*And that one was an impossibility*".

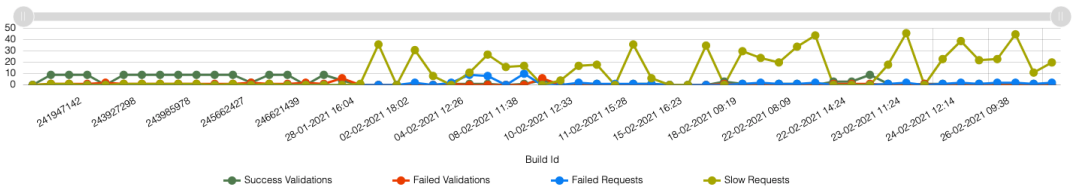
"*We would have found it before.*"



Example of AI Blueprint Dashboard

Each new blueprint will add data to the baseline (learning at each build) and set itself up for success for the next build. And if accessors change, it adapts to those changes (no maintenance required) as well as changes in user flow, reporting variances to a QA team as desired. As the team dug into the results, they were able to reproduce each error on their own. AIQ hands them the scripts which it wrote that caused each bug. They can be run in the IDE right on a developer's desktop. Each bug was verified to be genuine, and each was addressed and resolved within weeks.

One noteworthy bug occurred when a UX action caused the generation of almost 200 API calls, with all but one being returned. This client-side call was related to a UK-based server. Despite the client's insistence that such a request was impossible to be in their code, the machine observed it. And the machine doesn't lie. Of course, they verified this for themselves by running the autonomously generated script on a desktop computer. As it turned out, that server was shut down three years ago, but some shared client-side code continued to request and never received a response, resulting in a somewhat erroneous page. Despite this, none of their tests or testers ever encountered it, or if they did, they failed to pinpoint the main cause. However, the AI system accomplished precisely that. We discovered the root cause on day one. We did not use the server logs to generate more precise use cases in this situation, although we did later discover several additional concerns.



Example of tracking AI test results over time

Summarizing

There are many objections upfront to deploying such a system including We don't need it, It's not better than what we do, How can AI know our exact important business flows, I don't believe AI can find bugs our team can't.

I can easily (and deeply if we had another few thousand words) answer these early objections. But I would summarize in this way…ANY system which can augment our teams bug finding in less time with limited oversight is a very good thing indeed.

AIQ is not meant to be a replacement to QA teams. It's meant to augment your work and add to your skills as a team, not take any away. In time teams trust and rely on the results. AIQ can and will find most bugs long before your users do, can create and run scripts 100,000X faster than humans alone, and can see UI to API issues clearer than humans can.

As we move to become DevOps teams, we must lean into AI autonomous testing to generate more scripts and find more bug in our very precious short release cycles.

We all win when we deliver cleaner releases. And it makes our job frankly much more fun.

About the author

Kevin is CTO and Chair of Appvance.ai, a leader in AI based autonomous testing. He has been awarded 93 worldwide patents.

https://en.wikipedia.org/wiki/Kevin_Surace
Twitter: https://twitter.com/kevinsurace
Linkedin: https://www.linkedin.com/in/ksurace/

About Appvance

Appvance.ai is the leader in AI-driven test generation, which is revolutionizing how software testing is performed. The company's premier product is Appvance IQ, the world's first AI-driven, unified test automation system. It helps enterprises improve the quality, performance and security of their applications, while transforming the efficiency and output of testing teams. Appvance.ai is headquartered in Santa Clara, California, with offices in Costa Rica and India.

Visit us at
https://www.linkedin.com/company/appvance/
https://twitter.com/appvance
https://www.facebook.com/AppvanceInc/