

AI LOG INTERROGATION FOR AUTONOMOUS TEST GENERATION

BY KEVIN SURACE, APPVANCE

Key statement: An AI analysis of log file data can be used to re-create the UX actions from production users

Test automation today is difficult, time-consuming, and error-prone. It's difficult because it requires a highly skilled automation engineer to write, edit and maintain scripts. It's time-consuming because making sure those scripts work, and maintaining those scripts as the application changes, typically takes about a man-hour per script.

Additionally, application complexity continues to grow at a faster rate than test teams and tools can keep up with. Client/server computing has paved the way for service-oriented architecture and web services, which in turn have paved the way for web applications and N-tier computing, which are in the process of paving the way for microservices.

Further, each of these application architectures is a synthesis of everything that came before. Few applications adhere to a rigid architecture, preferring to reuse code and, in some cases, entire systems from previous applications.

Then there's the broader question of whether the automation scripts accurately reflect what users do in the application. While testers may draw on user personas and define a prospective workflow, both are educated guesses about the user's nature and the path through the application's goals and activities. Real users will not interact with an application in the manner in which test scripts are written.

To be effective, automated testing must accurately represent how real users interact with the application.

Today, test scripts are typically recorded or written by testers as they navigate through the application's multiple steps. Often, these steps are intended to test specific pages and functions rather than to replicate a user workflow. That is a good start, but errors frequently manifest themselves only when tasks are performed in a specific order.

Newer development methodologies are also exposing the limitations of test automation. Both agile methodologies and the newer DevOps trends demand fast and high-quality automation to keep up with an ever-increasing pace of rapid development and continuous deployment. The technical difficulty of automation today, coupled with the need to constantly modify test scripts to keep up with the software, prevents testing from being as responsive, fast and thorough as it needs to be.

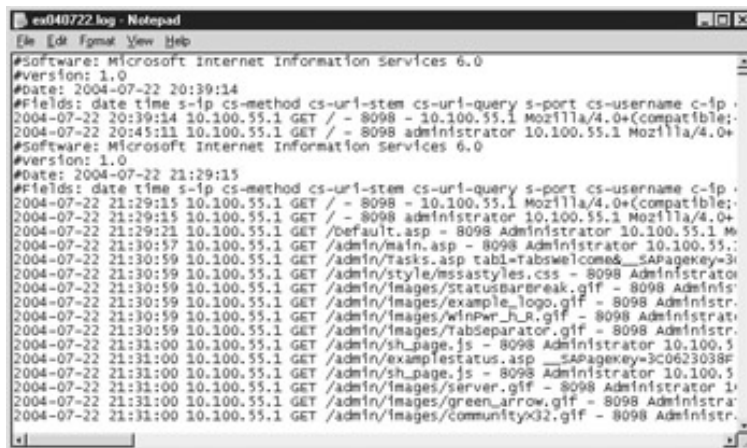
As a result, automation, despite its potential for improving quality and speeding up test execution, frequently falls short of that potential due to limitations inherent in existing automation tools and techniques, the majority of which are based on 30-year-old methodologies. Many organizations that attempt test automation frequently abandon it or relegate it to a minor role after discovering that it is not a silver bullet. Those who are successful with test automation invest a significant amount of human capital in it.

To be effective, automated testing must accurately model how real users interact with the application, as well as the capability to rapidly generate test cases based on those steps. Additionally, these test cases must be easily maintainable and adaptable to changes in the application. Representing actual use improves software quality by focusing on the workflows that matter.

Enter AI and Machine Learning

Which test suite is optimal for a new build? That is, in an ideal world? With an infinite supply of time, money, and people? It would involve logically applying user flows from production to the new build, along with any new feature user flows and validations, and observing the result. If we could accomplish this, we could create the ideal regression test for each new build. Observing results through the eyes of real users in a previous release.

The standard W3C log format (and related IIS, NCSA, Splunk, Sumologic and others) provides insights (most for security) to server activity in production. Log files may span days or weeks and be larger than 50GB.

A screenshot of a Notepad window titled 'ex040722.log - Notepad'. The window displays two blocks of W3C log format text. The first block starts with '#Software: Microsoft Internet Information Services 6.0' and shows a GET request from 8098 to 10.100.55.1. The second block starts with '#Date: 2004-07-22 21:29:15' and shows a series of GET requests for various files like 'example_logo.gif', 'winPer_f_r.gif', and 'sh_page.js'.

W3C format example (from flylib.com)

The following fields are recorded by the log file:

Field	Description
Remote hostname or IP address	The IP address of the remote user, or the hostname if DNS is available to resolve the name
User name	The remote login name of the user
Authenticated name	The username used to authenticate on the server, as with password-protected pages
Date	The date, time, and GMT offset of the request
Request	The method, URI stem, and protocol used for the query
HTTP status code	Records the protocol status message (such as 404, for HTTP not found)
Bytes transferred	The bytes transferred between the client and the server

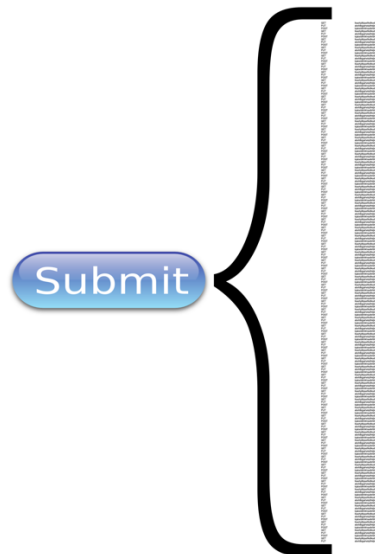
The production logs, as shown above, provide solid (and enormous) data showing requests made of the server. This alone is not enough to create much of anything (from a test perspective) since we have no binary information nor any context such as page or page state or even how this request was initiated.

Some background on server requests of note which is that a user action, on average, will make 200 requests of the server to fill the next page or screen. There are exceptions, such as same page

applications, where requests may be limited to AJAX or single requests for data. But even in single page applications data is requested from the server upon user actions in most or all cases.

So how can we get context? We could first create an application blueprint (formerly called a Master Key File) that would hold the “keys” to every single possible user action on every possible page state, and their resulting server requests. This would be a very smart AI system of robots which can navigate an application logically, completing fields and marching forward, capturing every action and page and state and server requests.

For example, let’s assume a simple submit button. By submitting a form with specific data, that data will be sent to the server and a variety of requests will be made of the server to deliver certain new elements and results based on that data. Here we see 200 requests made when that submit button is selected by a user:

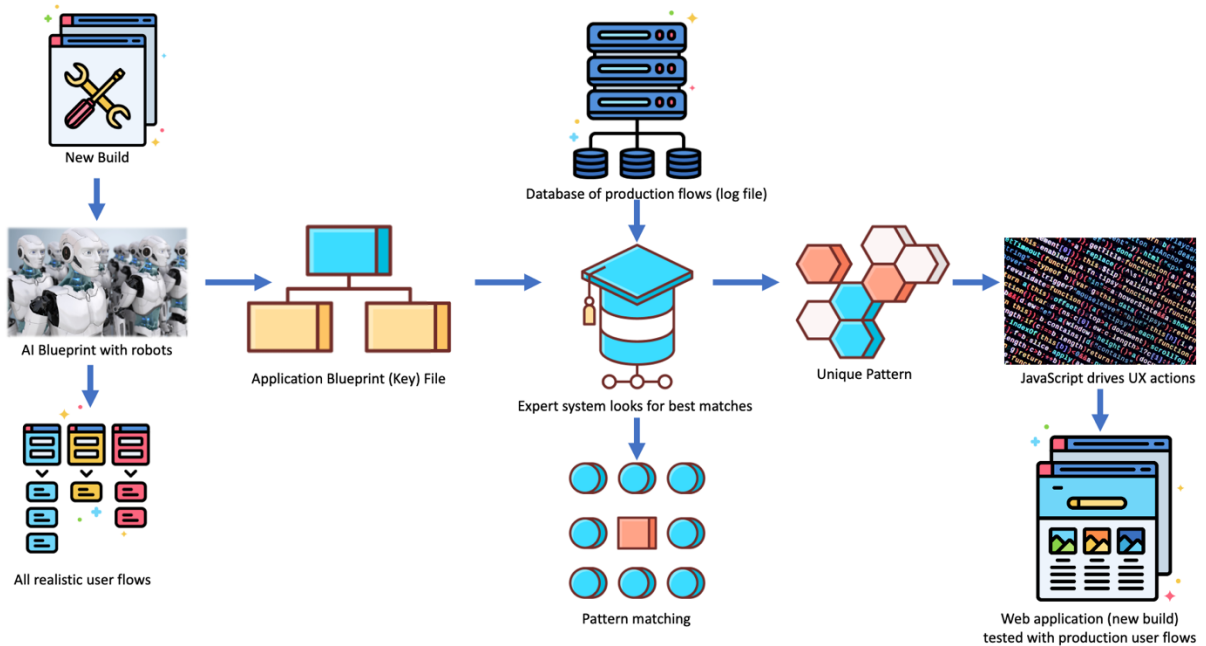


This example is a great way visually to understand what 200 unique requests looks like compared to that one user action. That set of requests turns out to be quite unique from other actions on this page. And we can use this fact to help us determine what UX flows a user has taken.

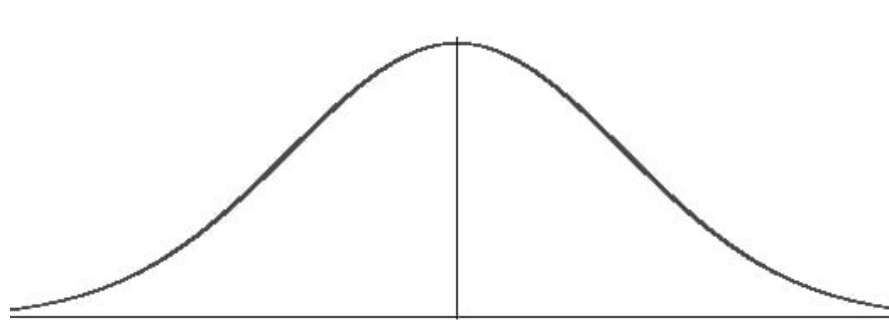
By recording every possible UX action in the AI driven blueprint, we group these requests into groups associated with unique specific actions on page states.

Later, we can begin interrogating the log files using an AI computational technique called predator prey. The goal here is that a log file has almost unlimited possibilities of gets puts posts and other requests recorded from users’ production interactions. It’s not atypical for a log file to be 50GB or more, representing 10’s of thousands of user flows and perhaps millions of server requests. As a result, the problem of locating a similar grouping that corresponds to the groups visible in the new build-related blueprint file appears to be infinite. Thus, a technique that is constantly looking for the top three candidates for a match and rapidly discarding the poor matches is extremely useful. Once we have found the best match (there may be no perfect match since the production build is slightly different than the

new build), the API group is matched to a UX action in the new build. This is fed to a code generator to generate JavaScript which will execute a UX action which corresponds to the closest match API group for that page state. Typically, we can create 5,000 scripts mimicking production user flows in about 10 minutes. And a QA engineer may decide to set various limits on this such as 1,000 scripts or even 100,000 scripts.



The “AI” here is really the learning which occurred by learning the application through the AI blueprint process, finding similar groups in the logs, and upscaling those API requests to corresponding UX actions in script form. The learning from AI blueprint is also retained for updating the baseline, highlighting differences between builds and so on. Thus, the machines learning is ongoing and forever, but targeted solely at your application.



The result is most often a bell curve of usage (that is hundreds or thousands of scripts) gleaned from production activity which are then applied to the new build.

In addition to the scripts, we will want to be able to validate other outcomes beyond working flows. This is also a learning process by leveraging these user flows with relevant test data to attain outcomes from these scripts against production. In this case, the AI blueprint can be used from a past run (the production build) to be sure and generate the correct scripts for production or simply use those scripts from that build already generated by the system. Then outcomes can also be recorded and reused against the new build to be certain that outcomes are validated as well as flows working perfectly.

This methodology is more secure and accepted than adding trackers to applications which can create serious security concerns and disclosure requirements.

This method has been successfully used across hundreds of applications, validating flows and outcomes as well as all API responses from the server side. Without a single script written by a human.

As the system learns from each build, there is no maintenance of scripts. There is no concern about accessors because the system will automatically apply new accessors to new scripts in each new build. There is no concern that important user flows are skipped because all important flows must be present in production (save new features). And there is no security concern because nothing is added to the application to accomplish this.

Many more details on exactly how all this works can be found in Appvance's patent [10,204,035](#).

SUMMARY:

By learning from both the new build and the previous production release (using AI) and combining this information with standard log files, a QA team can generate the bell curve of user flows that mimic those of real production users and apply those tests to the new build using an AI system such as AppvanceIQ. All of this is accomplished without writing a single line of code. Appvance first made this technology available in 2017.

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.appvance.ai>

<https://www.linkedin.com/company/appvance/>

<https://twitter.com/appvance>

<https://www.facebook.com/AppvanceInc/>