Of Machines and Men

Iryna Suprun

iryna.suprun@gmail.com

Abstract

Artificial Intelligence (AI) and Machine Learning (ML) are everywhere. Smart thermostats, self-driving cars and virtual assistants are changing our lives. AI and ML help people to make decisions and process information. They are used in medicine, education, retail, advertising and in many other industries, including software development and testing. Some even predict that soon AI will replace software programmers and testers. Nobody knows if these predictions will become true and if so when. There is no doubt although that AI will impact the way we test and will require a new set of skills.

Testing tools that use AI and ML are emerging and become more mature. Ready-to-use AI solutions are available on the market today and can be integrated into the testing process now. Paid and free AI and ML classes are offered by most of the online learning platforms.

In this paper, we are going to overview the state of AI in software testing and how it impacts the software development industry. Finally, we will take a brief look at AI-powered testing tools and discuss how we can use them to enhance testing.

Biography

Iryna started her career as a software engineer in 2004 in Ukraine, where she was born. She received her master's degree in Computer Science in 2006, and in 2007 she got her first position as a quality analyst. She moved to the USA in 2008 and continued her career in Quality Assurance (QA), focusing on the telecom industry and testing real-time communication systems and products such as the audio platform for the GoToMeeting application. Three years ago, she decided it was time to try something new and moved to AdTech. She is presently working as a QA architect at Xandr. Her primary job responsibilities - building automation frameworks and implementing testing processes from scratch.

1. Introduction

During the last fifteen years, QA as a profession went through many transformations to adapt to changes in technologies and in the software development process. Many predicted that QA as a profession would be eliminated by switching to agile, test automation, Test Driven Development (TDD) and Behavior Driven Development (BDD). It did not happen. Now there are concerns that AI/ML will succeed where others failed.

The most difficult task for Quality Analysts (QA) since the first test was written is executing and automating end-to-end tests. End-to-end tests (sometimes referred as system level tests) are tests that validate application flows from start to finish simulating real world scenarios. They validate the system under test and its components for integration and data integrity. They include communication of the application with hardware, networks, databases and other applications.

Even for a single application where all components are designed to work together, automation of end-to-end scenarios is a challenging, time and resources consuming task. These tests are often very complex and require a lot of maintenance, especially if the application is new and grows rapidly.

Automating end-to-end tests for a software platform where all components were designed independently, using different technologies and implemented by multiple teams, working in many locations and time zones, is a task that requires, based on my experience, quadruple time and effort investment due to increased complexity

The platform end-to-end flows are difficult to execute even manually, because they require complex thoughtful data setup, deep understanding of the platform, and knowing technical details. Add limited QA resources and challenging deadlines, and you have the problem I needed to solve.

The software platform for which I need end-to-end automation for consists of three UIs, over 50 APIs, data processing pipelines (queues, databases), and has multiple dependencies on external data. Teams are located in five geographical locations and four time zones. We mastered testing individual components using unit tests, traditional testing tools (Cypress and home-grown tool built form Cucumber and Java Script), mocks and simulators for external dependencies. Platform level end-to-end tests were our pain point and I needed a way to automate them as fast as possible.

Most AI test automation tools claim that users can automate the very complex scenarios without a single line of code, so the test automation process is really fast and can be easily done by anyone in the team.

Another big promise is the low maintenance. By collecting and smartly using numerous data points about flows, UI elements and user behavior, AI tools claim they are able to update tests without human interaction.

Will these two AI tools features, and others, enable me to create much needed end-to-end tests in days (instead of weeks) and decrease maintenance efforts at the same time? That's the question I wanted to find an answer.

There are many AI testing tools. The maturity of these tools, the usability, and the promises delivered vary. I reviewed the state of the AI-based tools market and tried some of the tools to find answers to two important questions: are tools that use AI as good as their marketing materials claim, and will human QA teams be replaced by these AI testing tools?

2. State of AI in Testing

There is no standard definition of what AI is. In general AI is software that can mimic human intelligence. Based on this definition, even a simple *if-else* statement can be considered as AI. Usually when people talk about AI today, they are really talking about ML algorithms that use large amounts of data to learn how to perform complex tasks. Hence, for the future discussion let's agree that AI-based testing means a tool that uses ML algorithms to solve one or another testing related task.

There are four big groups of ML algorithms: Supervised Learning; Unsupervised Learning, Reinforcement Learning, and Deep Learning.

- *Supervised Learning* is based on providing a large number of examples with known output.
- *Unsupervised Learning* is used to find patterns in the datasets, where outcome is not known.
- *Reinforcement Learning* and *Deep Learning* behave more like humans and use neural networks algorithms to be trained to solve specific tasks.

There are six (some say four) levels of Test Automation Autonomy.

- **Level 0** is manual testing. Even when everybody aims to automate as much as possible, it is around 70% of all testing[1] (this percentage varies from 50% to 85% and depends on the source and calculation methods). No AI/ML used.

- **Level 1** is traditional scripting and automated tests. It was introduced more than 20 years ago and still only 25% of testing. No AI/ML used.

- **Level 2** is codeless script generation or recording. It is 5% of testing today.

- **Level 3** is self-healing scripts and bots. The Supervised Learning algorithms are often used to improve self-healing or the existing regression suite.

- **Level 4** is an automatic generation of scripts with no human intervention, scripting or recording. Unsupervised Learning algorithms can be used in this case. For example, for data-mining of production logs and automatic creation of new tests according to those logs.

- **Level 5** is fully self-generating tests that are able to validate complex systems and do so autonomously. Some AI-based automation tools claim that they provide instruments to achieve this level of autonomy. Reinforcement Learning and Deep Learning can be used in this case to generate automated tests based on data analyses of clicks, links and GUI elements.

The overall percentage of Level 3 – Level 5 testing is very low, but it slowly and steadily grows replacing Level 0 – Level 2 automation.

3. AI-based Test Automation Tools

3.1 AI-based test automation tools vs traditional testing tools

The most visible difference between traditional and AI-based tools from the start is in test authoring. It is a manual process when a traditional tool is used. It starts with defining test cases, validation and assertions and, finally, coding and testing that the test script actually works. Most AI-based test automation tools

[1] The percentage of automated, manual and self-generated script provided based on World Quality Report 2019-20, PractiTest State of Testing™ Report 2020, Appvance “AI Driven Test Automation. A Transformational Breakthrough for Software Development” e-book [12] [13] [14]

record an actual user (test engineer or customer) manually executing the test case as the method to create the automated tests. Some testing tools that do not use AI also provide recording as a way to create automated tests. The main differentiator in this case is that AI tools collect tens of data points during recording that later are processed and used for improving/generating automated tests and their stability. Other, more sophisticated, AI-based tools use different ML algorithms to auto generate assertions and tests using logs, collecting real user clicks, or just software under test itself.

The test maintenance is a common pain point when we use traditional automation tools; it is manual and laborious. Tools that use ML algorithms offer self-healing features where tests automatically are updated to reflect changes. AI-based tools automatically adapt to UI changes, such as xpath, tag or other attributes. They use data collected during test creation to identify the same element on the page. User still can decline test change and roll back to the previous test version.

AI tools are not mature or widely used compared to traditional testing tools, so if users encounter a problem, they most likely need to go to the tool vendors support team. There is less available information about AI tools in general: comparison charts, use cases, real reviews and experience of real users are hard to find. So, if you are considering adapting an AI tool, be ready to do a lot of leg work by yourself.

There are plenty of open source or free versions of traditional test tools. The open source AI-based test tools, as well as free tools, are rare.

There are plenty of traditional testing tools for every type of application, most of the AI tools only support automation of the Web applications. There are just a couple AI tools that provide automation support for mobile apps.

3.2 AI Automation tools market overview

Almost every modern test tool available on the market claims usage of AI to some degree. AI-based tools can be put in three main categories. First, intelligently designed tools that can help to solve some automation issues, but they do not use any ML algorithms. These tools have **AI only in promotion materials**. It does not mean that these tools cannot solve some testing challenges, or they are worse than AI based tools. It only means that there are no ML algorithms used in their code. These tools are not a subject of this paper, so they are excluded from the future discussion.

The second category is tools that use **AI/ML in a supporting role**. These tools are used to help QA to perform certain tasks that are hard to do manually due to the limitations caused by human nature, for example, tools that are used for visual testing. Another use-case is to perform testing that if done by people would use subjective measurements instead of a set of objective parameters (video/audio quality).

Tests authoring in such tools is done with active involvement of end-users (test generation based on collecting, extracting and processing logs, clicks and events) or members of the engineering team (test recording).

Last category of AI-based tools are ones that take **software under test** as their **main and only input** and generate bug reports as output without any human interactions (Level 5 of autonomy). There are no tools on the market right now that are Level 5 tools. There are some tools, not many although, that have Level 5 features, but most of their offering consists of Level 2 – Level 4 features.

As the majority of tools use AI/ML in a supporting role nowadays we will focus on them and discuss features that make them very solid competitors to traditional tools.

The first big group of such features, ones that help decrease time spent on tasks that people can accomplish and probably do better than any non-AI algorithms, including the following:

- Codeless script generation. Writing code for complex end-to-end scenarios can take as long as development of a new feature; with codeless script generations even non-technical people (for example, end-users during User Acceptance Testing) can record their actions and a smart tool will convert those into automated tests. An automation engineer most likely still will not be able just add these recordings to the test suites. There will be some work required, like setting up users, make object names to comply with naming conventions, etc. Nevertheless, it still can significantly decrease time spent on initial automation of test cases and expand coverage.

- Self-healing. Test case maintenance is probably the most hated task among the developers and QAs. It takes a lot of time; it slows down everybody. AI tools can collect a lot of information about every element on the webpage, so if one attribute is changed, the tool can still recognize this element and proceed with execution. Some tools also collect information on how applications are used, like user flows, errors, etc., and are able to recognize insignificant changes and adapt.

- Converting test documentation to automation. Using natural language processing mechanisms, it is possible to convert tests written in English to automated tests. Some tools advertise that they can do it with both structured test plans and unstructured user journeys.

The second group - features that perform tasks that are very challenging or almost impossible to do by human beings – including the following:

- Autonomous building of test cases based on usage traffic from real users. These applications collect analytics data from an application clickstream, analyze it and create tests cases based on real system usage. They identify core patterns (sequences) and then run them in the test environments to improve scripts using ML algorithms (remove not required steps, duplicated flows, etc.)

- Autonomous building of test cases based on the analyzing code of application under tests. Bots build the map of software by exploring each path through it and then create set of use cases based on it.

- Visual Testing. Visual testing evaluates the visible output of an application and compares that output against the results expected by design. Some may think that it is a task better performed by humans, but it is just time consuming and we often miss things that a computer would not. Often humans do not pay attention to the things they see many times a day, sometimes people don't see obvious differences (think spot 10 differences between two pictures), and finally, humans cannot go through all screens multiple times a day and notice every difference. Visual testing allows to test the whole UI. Using ML algorithms helps to decrease number of false positives by identifying changes that do not impact user experience and ignoring them.

- Audio/Video Quality Testing. Before it was a highly manual task that required listening to the audio or watching video and giving it a subjective score. Today AI can collect multiple data points about audio/video and how their variations impact audio/video perception by humans. They can perform testing faster based on this data, make it more objective and ignore variations that are not important for humans.

4. AI-based Testing Tools - Field Study

4.1 Promises Delivered

4.1.1 Quick Start

Recall that (1) automation of complex platform-level E2E tests in the shortest possible time and (2) decreased time spent on their maintenance were the two problems that led to this research. I read a lot of documentation, white papers and watched video lessons for many AI-based automation tools. You can find the comparison table of most popular and most promising AI-based tools in Appendix 1. I selected

three of them (Testim, Mabl, and TestCraft) and automated one complex test scenario using each of these tools.

As time was my most valuable resource, I concentrated mostly on the time-saving features. The first most pleasant discovery was how easy it is to start using these tools. All of them have easy to use web-interfaces and provide extensive documentation. It took me just a couple hours from opening a tool for the first time in my life to having my first automated test running. Of course, one should invest way more time to use any of these tools to full potential. Using advanced features like adding code snippets, reusing steps, adding variables, setting up test data, managing test suites and execution requires more training and practice.

4.1.2 Codeless Script Generation (recording)

The next feature I explored is codeless test generation. This feature is implemented in all tools I tried with different levels of maturity. I found that Mabl is the most mature one. Recording of the test using this tool was easy. It was able to handle some challenging navigation tasks, such as switching between different web-applications, finding and selecting checkboxes in dropdowns, and hovering.

Recording tests in TestCraft is slower and requires an extra step to record each action. This might be a showstopper if you plan to involve anyone outside the engineering team into test creation. The real advantage of test recording that anybody can do it. You can ask the product team to record the user acceptance tests, support team to record flows where end-users noticed issues, or even real users to record their action. QA engineers can modify these recorded flows and add them to the regression test suite later. It should not require anything except hitting the record button and forgetting about it to be able to engage non-technical part of the team or end-users in the test recording. If there is anything extra, chances are that they refuse doing it. Unfortunately, in TestCraft it is not possible. I also found that navigation between different modes (recording, execution, editing) is a bit confusing in this tool. On the other hand, I should mention that tests recorded in TestCraft were very stable from the get go and required very little editing after recording.

Test recording in Testim (community version) is least mature. I was able to record the tests easily but they failed in multiple places when I tried to re-run it immediately after recording. To be able to do such actions like switching between different applications, selecting an item from dropdown I needed to add some delays, re record specific action, etc. I did not face such challenges with other tools.

4.1.3 Decrease of maintenance time

It is hard to estimate the advertised decrease in maintenance time during the evaluation period, especially if one has a limited number of automated tests (which is usually the case for Proof of Concept). There is also a need to have the same tests automated using traditional tools that are used in parallel to be able to compare time spent on maintenance of the same tests.

Tests automated by me using different AI-based tools were executed on multiple builds and releases. Every tool I tried handled insignificant changes in the UI, such as text, size or other attributes well. If the change was something that cannot be handled by a tool (introduction of new elements, major redesign of UI) it was much easier and faster to fix automated testcase by re-recording steps than introducing the same change using code.

4.2 Challenges and obstacles

4.2.1 Codeless script generation

Although the creation of tests using recording features is much easier and faster than using traditional tools, it rarely went absolutely smooth. Sometimes I needed to re-record steps or part of the script. Sometimes I needed to manually add delays. Creating custom object names that follow naming convention, having dates in the names was a challenging task in most cases. I believe that when QA

looks at the data generated by automated tests she should be able to figure out what test created this data, when, and probably what build was used without opening logs or doing anything extra. There is always the possibility to add JavaScript code snippets (all tools I tried have this feature) to implement this or other custom behavior but in this case, it is not codeless anymore. If code is added all disadvantages of maintaining it come into place, including not easy debugging. As we see despite all achievements in the AI industry, recording tests is still a bumpy process for complex UIs and scenarios even today.

We also should remember that the ability to record tests does not magically solve all common issues of automation. It does not matter if you use code or recording to create tests you still need to make sure that you automate what matters. Tests should be well designed and automated at the correct level. One should also think about test data management, including setup and tear down.

Recording tests can speed up automation by providing an initial set of raw scenarios to work on. Still, automation engineers need to work on defining and reusing steps that are common for many tests, setting up and maintaining testing accounts, selecting test subsets to execute on different stages of the software development lifecycle. Recording all this is not a magic bullet. Engineers still should work to have robust, easy to maintain automation tests.

4.2.2 Self-healing feature is a double-sided sword.

Self-healing is a great feature, but it does not mean that all your tests will be magically fixed every time. Yes, insignificant changes are fixed automatically, but when your UI is mature, you don't have these changes introduced often. I would say such changes are rather rare after a couple of major releases. Definitely collecting multiple data points about each element and using them for element location improves stability of tests. It rather impacts robustness (tests do not fail if text on element changes depending on the time of day, for example) than decreases maintenance time(tests needed to be updated because of some changes in code). I found that self-healing saves way more time if the application has brand-new UI that changes often.

Different tools handle self-healing in different ways. Some of them require approval of every change, at least at the beginning while the system learns, so it is still a time investment. Others just make changes and proceed without notification, which I think is dangerous. For example, a currency sign in the platform I test and the number of digits after decimals is important but if they are missing - the self-healing systems will "auto-fix" such cases. Fortunately, most of the AI-based tools are starting to add the ability to review auto-fixed tests, accept or decline these changes, track history, and rollback to previous version. Don't forget that reviewing changes still requires time. I also suggest that you need to double check what is the mechanism of handling auto-changes before selecting one or another test tool with self-healing features.

To overcome the "auto-healing" problem for the small but important changes that should not be ignored, tests can always be built in a very specific way, where every sign, every comma has its own assertion. The result of this? Huge, slow and challenging to maintain automated tests. We can also make tests smaller and increase the number of tests, but if each test requires its own time-consuming setup having hundreds of such tests will lead to the increased time of test execution. We cannot wait hours each time we build the software.

Every big change in the application, like introducing a new element, removing tabs or anything else, will still result in the test updates that should be done manually. Of course, QAs will be able to do it faster because re-recording steps is faster than adding code. We should remember that it will be faster if QAs use any tool that provides the recording, not only AI-based tools.

4.2.3 Self-generated tests: improve the test coverage, what about quality?

The AI-based automation tools can generate tests in different ways. Some of them generate tests by collecting information on how real users use the software in production. This information can be extracted from logs, clickstream, or both. This probably works for software that uses the Business-to-Consumer model, then there are many users that produce a lot of data for ML algorithms. In the case of Business-to-Business model there are often less users and it is harder to collect enough data to train ML models.

There are also many applications/features that do not have “users”, for example, different reports that are generated as a result of data processing algorithms and calculations. There is another big drawback of generating tests in such a way – the application or the feature should be in production, and there should be users, so what about new functionality. That said, test generation based on the usage of application can only be used to add missing regression cases.

Another type of self-generated tests are tests generated by link crawlers. These tests check that every link in the app works. This is definitely a useful tool, but a working link does necessarily mean it’s the right link, or that its a functioning application.

The auto generated tests only cover what can be easily tested. They find shallow bugs, like not working buttons, particular values, broken links. They will never help you find the requirement that was missed, logic flaws, the error message that was not generated to help the user to overcome the problem, or spot usability issues. Moreover, auto generated tests might give you a false feel of security and allow major issues that escape to production after thousands of automated tests have passed. Such auto generated tests are great to supplement QA efforts, but not a full replacement.

Another concerning thing is that marketing materials for these tools claim that using one or another way of autonomous tests generation will give you 100% of coverage. This raises the question of 100% coverage of what and more important do you even need to achieve these mystical 100%.

5. Summary

The usage of AI and ML in the testing tools is increasing daily. Almost every tool claims that there are some AI powered features that help improve testing. The key word here is “help”.

The test recording was introduced a long time ago, but even introducing AI and ML did not make it perfect. It is much better than it was, it saves time, but it still requires human intervention, sometimes for simple cases.

Fully autonomous test generation although sounds cool, is not mature enough to leave all your testing in its hands. It can be useful to reveal gaps in coverage and easy to find bugs. Testing of complex systems cannot be done using only autonomously generated tests. These tests will never ask “what if”.

The same goes to self-healing tests. It allows QAs to save time and removes a lot of mechanical, mindless work, but it is not a magic bullet of a fully autonomous task. Users still need to review changes. Moreover, QA Engineers need to design tests using best test automation practices and techniques and incorporate self-healing into design, so actual issues are never auto fixed (add explicit asserts, for example)

I may sound old school but the test that passes even after a change was introduced sounds to me more concerning than exciting. Isn’t the purpose of the most automated tests to uncover changes that were accidentally introduced and else would go to production unnoticed? Why do we want to hide these changes?

The AI tools are great in the areas where a lot of information should be collected and quickly processed, like visual testing, video/audio quality, log parsing, collecting real usage information, performance testing. Visual testing was not covered much in this article only because they do not fit my particular use case, but there are many success stories across industry. It is impossible for a human to detect any single visual change in UI, so using software is justified and pays off. AI in this case helps to find only differences that are perceptible to end-users. They do not use simple pixel-to-pixel comparison, but instead they use class of AI algorithms called computer vision, that helps to keep signal to noise ratio high.

To have the robust, stable, effective, lightweight, trustworthy and easy to maintain automated tests, one should not only select the right tool but also invest in test design. Someone still needs to find out what should be tested, when and how to do it in the most efficient way. AI tools’ test designing abilities are

very limited, almost non-existent. Mechanical clicking on everything clickable and extracting test scenarios from logs or clickstream can hardly count as a test design technique. AI can automatically generate a lot of tests that can be easily skipped and do not bring any business value or improve quality. Running thousands of tests is either time-or resource consuming so it is still QAs job to find the right balance between increasing coverage and speed of execution.

Tests often require complex data setup in software platforms or just large applications and none of the tools I've seen have features that help solving this task. Data seeding, maintenance and cleanup still should be done with some external tools or workarounds. I found this task sometimes most challenging when we talk about testing software platforms, solutions and software that uses a business-to-business model.

Finally, to have an effective automation the software itself should be designed with testing in mind, and no tool will be effective if application is untestable.

To summarize, AI-based testing tools in the current state cannot fully replace QAs. They can be used in a supporting role and decrease the time spent on mechanical, boring tasks. They are great as a supplement to the QA team, to automate time consuming activities, such as checking that every link in application is working, or tasks that are impossible for a human, such as spotting every visual difference between current and previous build. AI-based tools can be used to collect data about users' behavior to generate production like test scenarios. They also provide the ability to generate tests from existing test documentation.

Existing AI-based testing tools are mainly aimed at web or mobile apps. Nevertheless, these tools continue to evolve and decrease human involvement into checking while at the same time increasing human productivity and giving them more time to concentrate on testing[15].

Will we lose our jobs? Well, any process automation results in the disappearance of one or another job function. Do we need to adapt? That is definitely so, but we already did it many times before, didn't we?

6. Appendix 1. AI tools comparison table

The table below is comparison table of five most popular and most promising, in my opinion, AI-based tests automation tools. The *"+"* indicates that tool has or claims to have one or another feature or capability but does not indicate the maturity of that specific feature. The information for TestCraft, Testim and Mable provided bases on hands-on experience and research. Information about Appvance and Functionize features provided based on research only.

		Testim	Mabl	Appvance	Functionize	TestCraft
1	Codeless Script Generation/Recording	+	+	+	+	+
2	Autonomously Generated Scripts	-	-	+	+	-
3	Visual Testing	*Integration with Applitools and Test Rail	+	-	+	
4	Auto-healing	+	+	+	+	+
5	Mobile Testing	-	-	+	+	-
6	Code Snippets	+	+	+	+	+
7	Test Plans written in Simple English to Automated tests	-	-	-	+	-
8	Support of cross browser tests	*Integration with Browsestack, SauseLabs	+	+	+	+
9	Performance and Load Testing	-	-	+	+	-
10	Security Testing	-	-	+	-	-
11	API steps	+	+	+	+	+
12	Version Control Systems (VCS)	GitHub Bitbucket	GitHub Bitbucket	Git repositories	Github	Gitlab
13	CI tools	Jenkins Circeci TeamCity Travis Ci Codeship	Bamboo Jenkins CodeShip Azure Pipelines	Circeci Bamboo Hudson Jenkins	Bamboo CircleCI Jenkins Travis CI Spinnaker Go AWS CodePipeline Heroku TeamCity	Jenkins Visual Studio Team Services TeamSity
14	Collaboration Tools	Trello Jira Github issues Slack Email	Jira Slack	Rally Chef	Jira PagerDuty	Jira Slack
15	Big Data		BigQuery			
16	IDE	JetBrains, Visual Studio				

7. References

Web Sites:

[1] Nick Health. What is AI? Everything you need to know about Artificial Intelligence. https://www.zdnet.com/article/what-is-ai-everything-you-need-to-know-about-artificial-intelligence (accessed June 26, 2020).

[2] James Le. A Gentle Introduction to Neural Networks for Machine Learning. https://www.codementor.io/@james_aka_yale/a-gentle-introduction-to-neural-networks-for-machine-learning-hkijvz7lp (accessed June 26, 2020).

[3] Introduction to Artificial Intelligence (AI) Microsoft – DAT263x. https://www.edx.org/course/introduction-to-artificial-intelligence-ai-2 (accessed Jan 9, 2020).

[4] Testim https://www.testim.io/

[5] Mabl https://www.mabl.com/

[6] Appvance https://www.appvance.ai/appvance-iq

[7] Functionize https://www.functionize.com/

[8] TestCraft https://www.testcraft.io/

[9] Vu Nguyen. Test Case Point Analysis http://www.qasymphony.com/media/2012/01/Test-Case-Point-Analysis.pdf (accessed July 20, 2020).

[10] Javier Ferrer, Francisco Chicano and Enrique Alba. Measuring Testing Complexity University of Mlaga, Spain http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.188.6343&rep=rep1&type=pdf (accessed July 20, 2020).

[11] Michael Bolton. Blog: It's Not About The Typing https://www.developsense.com/blog/2020/08/its-not-about-the-typing/ (accessed August 15, 2020)

[12] World Quality Report 2019-20. Quality underpins the key business drivers of every major enterprise https://www.microfocus.com/en-us/assets/application-delivery-management/world-quality-report-2019-2020

[13] State of Testing™ Report 2020 https://www.practitest.com/resource/state-of-testing-report-2020/

[14] AI Driven Test Automation
A Transformational Breakthrough for Software Development
https://www.appvance.ai/portfolio-post/ebook-ai

[15] Michael Bolton. Testing vs. Checking https://www.developsense.com/blog/2009/08/testing-vs-checking/